|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分类号 |  | |  | | | 密 级 | |  |
| U D C |  | |  | | | 编 号 | | 10486 |
|  | | | | | | | | |
| 武汉大学logo  **硕 士 专 业 学 位 论 文** | | | | | | | | |
|  | | | | | | | | |
| 分布式服务治理框架的原理与实现 | | | | | | | | |
|  | | 研究生姓名 | | ： | 王伟 | |  | |
| 学号 | | ： | 2014212113103 | |
| 指导教师姓名、职称 | | ： | 龚奕利 副教授 | |
| 专业类别（领域） | | ： | 计算机技术 | |
|  | |  |  | |
|  | |  |  | |
|  | | | | | | | | |
| 二〇一七年五月 | | | | | | | | |

Dissertation Submitted to Wuhan University

For Degree of Master Engineering

The principles and implementation of the distributed service governance framework

By

Wang Wei

Under the Guidance of

Professor Gong Yi Li

May，2017

**郑 重 声 明**

本人的学位论文是在导师指导下独立撰写并完成的，学位论文没有剽窃、抄袭、造假等违反学术道德、学术规范和侵权行为，否则，本人愿意承担由此而产生的法律责任和法律后果，特此郑重声明。

学位论文作者（签名）：

年 月 日

# 摘 要

应用业务是不断发展变化的，业务创新越来越成为企业核心竞争力的一部分。传统IT支撑业务发展的方式是为新业务构建新的业务系统，系统多是“竖井状”的孤岛形式。久而久之，这样的方式已经无法满足业务发展对IT带来的要求，IT成 为企业的严重短板。面向服务的架构的出现的主要目的就是彻底改变这种局面，使IT变得灵活，与业务对齐。甚至能够促进业务的发展，并创造新的业务价值。本文研究了国内外的服务治理框架框架，分析和总结了通用的框架特性和非功能特性，实现了一号店的服务治理框架Hedwig。

首先，在服务治理概述中论述了服务化架构的演进，描述了服务治理的需求，阐述了服务框架的整体架构和技术选型。

其次,框架的特性包括功能特性和非功能特性。功能特性表示框架是什么，有哪些功能模块和组件。模块是细粒度的功能，组件是粗粒度的功能。后续章节给出了每个模块和组件实现的原理和设计方案。

最后，对于服务治理框架的容错和监控寻错功能进行了研究。故障是分布式系统要面对的主要困难之一，非功能特性中的容错模式为框架提出了提高服务可靠性和可用性的方式，在此基础上设计了hedwig的容错方案。服务监控系统Detector对异常服务调用的排查手段加强了整个治理平台的查错排错能力。

从功能上看，Hedwig已然是功能完善的服务框架，能给团队提供统一和快速的服务层接入。从非功能特性上看，hedwig为公司的业务团队提供了一个支持并发，吞吐量，容错可靠性，可用性都很高的分布式服务框架。通过将本文所研究和设计的系统hedwig在一号店全公司的推行，在这5年间hedwig已经满足了绝大部分团队的功能需求和性能需求，简单快速的接入方式也极大的节约了公司的成本。

关键词: 服务化治理；分布式服务化框架；分布式服务模式；容错模式

# Abstract

The application business is constantly evolving, and business innovation is becoming a part of the core competence of the enterprise. The way that traditional IT supports business development is to build a new business system for the new business, which is mostly an isolated form of "silo". Over time, this approach has failed to meet the demands of business development for IT, which is a serious shortboard for businesses. The main purpose of a service-oriented architecture is to completely change the situation and make IT flexible and aligned with the business. It can even promote business development and create new business value. This paper studies the framework of service governance framework both at home and abroad, analyzes and summarizes the general features of the framework and the function characteristic, implements the Hedwig number one shop service governance framework.

First, this paper discusses the architecture of service in the service management overview evolution, describes the demand of service management, expounds the overall architecture of the service framework and technology selection.

Second, the features of the framework include both functional and non-functional features. The feature section describes what the framework is and what the functional modules and components are. Modules are fine-grained functions, and components are coarse-grained functions. The following sections give you the principles and designs for each module and component implementation.

Finally, the fault tolerance of service governance framework and the monitoring and fault finding function were studied. Fault is one of the main difficulties faced a distributed system, the features of fault tolerant model for the framework puts forward the ways that improve service reliability and availability, on the basis of the hedwig fault-tolerant scheme is designed. The monitoring system of the service monitoring system has enhanced the search and error capability of the whole governance platform.

In terms of functionality, Hedwig is already a fully functional service framework that provides a unified and rapid service layer access to the team. Look from the non-functional properties, the hedwig team provides a support for the company's business concurrently, throughput, reliability and fault tolerance of high availability are distributed service framework. Through this research and design of the system the hedwig in one shop all the implementation of the company, in the five years the hedwig already satisfied most of the team's functional requirements and performance requirements, simple and quick way of accessing and greatly saves the cost of the company.

Key word：Service governance**;** Distributed service framework**;** Distributed service pattern; Fault-tolerant pattern

**目 录**

[**摘 要** I](#_Toc483557593)

[**Abstract** II](#_Toc483557594)

[**1 引言** 1](#_Toc483557595)

[1.1 研究问题的背景 1](#_Toc483557596)

[1.2 国内外研究现状 2](#_Toc483557597)

[1.3 研究的内容与方法 2](#_Toc483557598)

[1.4 概念和名称 4](#_Toc483557599)

[1.5 论文组织结构 4](#_Toc483557600)

[**2 服务治理概述和框架相关技术** 5](#_Toc483557601)

[2.1 服务化背景 5](#_Toc483557602)

[2.2 服务治理需求和分析 6](#_Toc483557603)

[2.3服务框架中的角色 9](#_Toc483557604)

[2.4 服务框架技术选型 10](#_Toc483557605)

[2.5 本章小节 11](#_Toc483557606)

[**3 服务框架功能特性** 12](#_Toc483557607)

[3.1 服务注册和发现 12](#_Toc483557608)

[3.2 服务路由和负载均衡 14](#_Toc483557609)

[3.3 序列化和反序列化 15](#_Toc483557610)

[3.4 服务调用 16](#_Toc483557611)

[3.5 RPC通信框架 18](#_Toc483557612)

[3.6 服务发布和引用 19](#_Toc483557613)

[3.7 服务框架部署方式 21](#_Toc483557614)

[3.8 协议栈 22](#_Toc483557615)

[3.9 幂等 23](#_Toc483557616)

[3.10 本章小节 24](#_Toc483557617)

[**4 服务框架功能组件和模式 25**](#_Toc483557618)

[4.1 客户端代理 26](#_Toc483557619)

[4.2 请求者 26](#_Toc483557620)

[4.3 客户端请求处理器 26](#_Toc483557621)

[4.4 调用者 27](#_Toc483557622)

[4.5 服务端请求处理器 28](#_Toc483557623)

[4.6 接口描述 28](#_Toc483557624)

[4.7 远程错误 28](#_Toc483557625)

[4.8 本章小节 29](#_Toc483557626)

[**5 服务框架容错原理和模式** 30](#_Toc483557627)

[5.1 集群容错 30](#_Toc483557628)

[5.2 服务降级 30](#_Toc483557629)

[5.3 服务可靠性 32](#_Toc483557630)

[5.4 服务隔离 33](#_Toc483557631)

[5.5 流量控制 35](#_Toc483557632)

[5.6 超时与重试机制 38](#_Toc483557633)

[5.7 服务容错模式 40](#_Toc483557634)

[5.8 本章小节 42](#_Toc483557635)

[**6 Hedwig服务框架设计和实现** 43](#_Toc483557636)

[6.1 Hedwig的模块和时序图 43](#_Toc483557637)

[6.2 Hedwig的功能模块设计和实现 45](#_Toc483557638)

[6.3 Hedwig功能组件设计 52](#_Toc483557639)

[6.4 hedwig容错设计和实现 53](#_Toc483557640)

[6.5 Hedwig测试与性能 59](#_Toc483557641)

[6.6 本章小节 60](#_Toc483557642)

[**7 服务监控和治理** 61](#_Toc483557643)

[7.1 监控的种类 61](#_Toc483557644)

[7.2 Hedwig监控 61](#_Toc483557645)

[7.3服务监控上的应用 64](#_Toc483557646)

[7.4 本章小节 65](#_Toc483557647)

[**8 总结和展望** 67](#_Toc483557648)

[**参 考 文 献** 68](#_Toc483557649)

[**致 谢** 70](#_Toc483557650)

# 1 引言

### 1.1 研究问题的背景

近年来，各大互联网公司都遇到了快速业务发展的情况,访问流量也是日益递增。因此，设计一个高可用、高性能且伸缩性强的网站架构逐步成为互联网领域关注的重点。经过研究发现，网站架构已经逐渐从集中式向分布式应用架构转变,如图1.1所示。并且，当今大型网站基本上是分布式设计的。

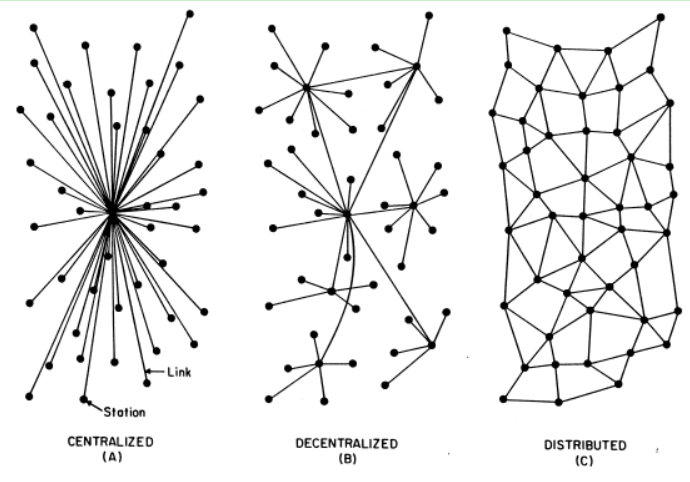


图 1.1 网站架构演进

分布式应用架构的核心思想之一是通过增加服务层来使远程调用服务透明化。这样也可以减少开发和维护成本，并使用户有一个统一的编程模型。分布式服务化治理框架就是引入服务层的重要方法。

目前，在互联网公司中，成熟的服务治理框架已经非常多，但大多没有开源。但也有少部分优秀的已经开源，并且被很多互联网公司直接使用。各家公司的服务框架都有自己的实现方式，不一而足,各有千秋。

服务层的上层是各种应用，承载了公司几乎所有的业务， 所以服务的稳定可靠性和性能，也会直接影响到电商用户的购物体验。他山之石可以攻玉，借鉴优秀的开源框架并结合公司自身的特点，设计并实现一个通用分布式服务治理框架具有重要意义。

### 1.2 国内外研究现状

如今,在一定规模的互联网公司，都会有基于服务化理念的产品, 其本身也是基础架构中重要的组成部分。

从2010年开始， 国内各个互联网公司都开始进行了整体服务化的进程，其中沉淀了不少优秀的服务化治理中间件。开源的包括淘宝的Dubbo，美团大众点评的Pigeon，网易的Motan，非开源的包括唯品会的OSP，一号店的Hedwig， 京东的JSF。

国外服务框架以Facebook开源的Thrift和Twitter开源的finagle最为受欢迎，thrift本身是个rpc框架，有很好的平台中立性， 国内很多互联网公司在thrift框架之上开发了自己的服务框架，其中包括唯品会的服务框架OSP。

官网显示，2012年淘宝的Dubbo有2,000+个服务每天承载30多亿次的服务调用。

Dubbo的部署方式更轻量， Dubbo可运行在任何Java环境中，而HSF的侵入性大。这方面Hedwig的akka版本也是轻量级的，框架本身也是容器，可以在有jvm的机器上快速部署。

Dubbo扩展性极好，很方便二次开发。Dubbo使用微核+插件体系的方式来做扩展，即所有功能都可以在不修改原生代码的情况下在外围扩展。这个是很少框架能做到的，也是hedwig需要学习的地方。

Dubbo的功能很全，比如有ClassLoader隔离，这方面hedwig是需要学习的，争取完善自己的功能。

美团大众点评的Pigeon也是很优秀的服务框架，功能也是比较全的，大部分的功能hedwig都是有的。Pegion有个创新的点，客户端路由策略，提供服务预热功能，能解决线上流量大的service重启时大量超时的问题。

国外的thrift主要提供了IDL(interface description language),这是相对中立的语言，所以可以支持多种语言，国内有唯品会的OSP和点评的pigeon使用了thrift。

### 1.3 研究的内容与方法

本文的主要研究目标是设计和实现服务框架Hedwig来为业务团队提供统一的服务接入方式，以此来降低应用开发的成本。从整体上看，服务框架可以分为功能性特性和非功能性特性。功能特性表示框架是什么，是做什么的。组件是粗粒度的功能，模块是细粒度的功能。框架可以由几个粗粒度的组件组合在一起，并且可以把这些组件的职责抽象到模式。框架也可以由许多细粒度的模块组合在一起。 组件是粗粒度的，组件会包括细粒度的模块。

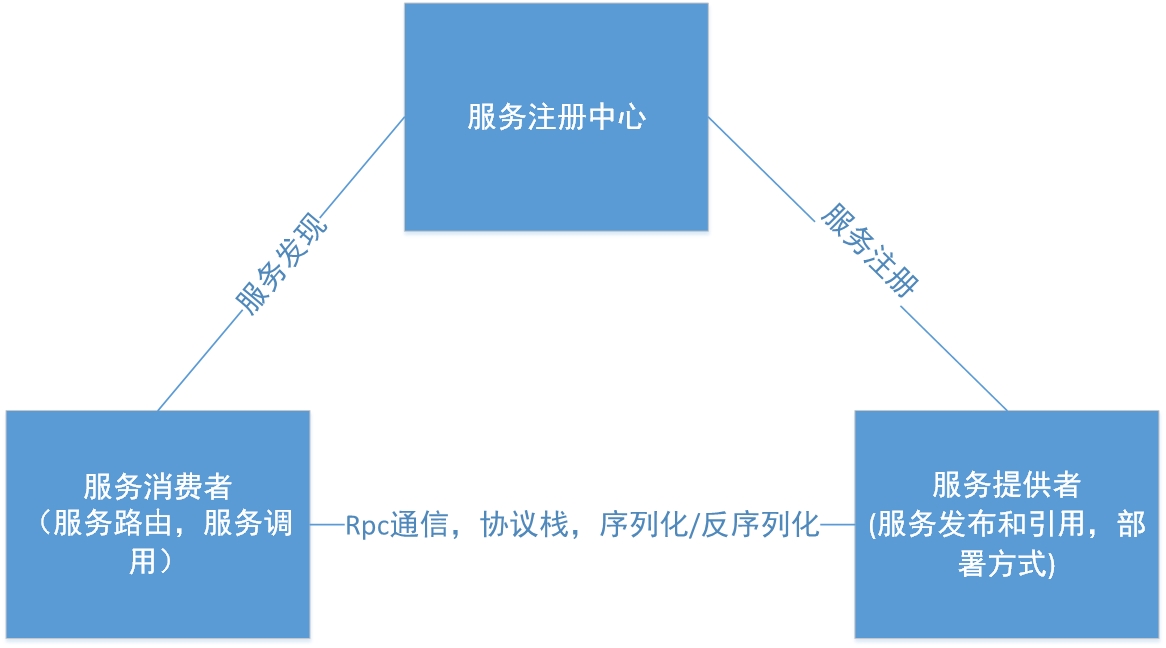


图 1.2 服务框架功能

研究的Hedwig功能性特性包括以下几点：服务注册和发现，服务路由(软负载均衡), 序列化和反序列化，服务调用，rpc通信框架，服务发布和引用, 部署方式, 幂等。这些特性和服务消费者、服务提供者和服务注册中心三者的关系，如图1.2所示。其中服务路由、服务调用主要是消费端实现的功能。服务发布和引用，部署方式主要是服务提供者实现的功能。rpc通信框架，协议栈，序列化和反序列化，服务注册和发现是消费者和提供者两边共同实现的功能。

Hedwig非功能特性研究的是系统提供的容错可靠性，稳定性，性能方面的问题。网络本身是不可靠的，远程接口可能响应慢或者宕机。根据“Design for failure为了错误而做设计”的理念，容错模式提供了通用的方案，为服务框架提供了很好的参考。

在hedwig运行时背后有一个服务监控治理中心，本身也是个平台，叫做Detector。利益相关人员通过Detector对服务调用的历史，当前的运行状态，健康度和服务依赖关系等各个指标维护和分析，能够在符合当前高质量运行的情况下，持续优化服务,或者能在服务调用有异常的时候排查问题。

### 1.4 概念和名称

定义：

Domain是一号店的部门;

Pool是一号店的子部门;

App是一号店包含业务的应用;

Hedwig 是一号店分布式服务框架;

Detector是一号店分布式服务监控和治理平台;

Hystrix是Netflix的服务容错框架;

Dubbo是淘宝服务治理框架;

Pigeon是大众点评服务治理框架.

### 1.5 论文组织结构

本文余下章节的组织结构如下：

第二章节主要是概述一下服务治理和介绍服务框架的相关技术。第三章节主要是介绍一下服务框架功能模块。第四章节主要是介绍服务框架功能组件和组件模式。第五章节主要是介绍服务框架非功能特性,包括容错原理和模式。第六章主要是介绍Hedwig服务框架详细设计和实现。第七章主要是介绍服务监控和hedwig监控治理平台，关注服务运行健康度。第八章是总结与展望，主要总结一下现阶段工作的内容和不足，展望一下未来研究的重点。

# 2 服务治理概述和框架相关技术

### 2.1 服务化背景

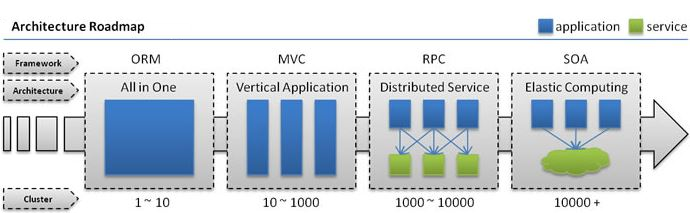
随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及弹性计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。

图2.1 面向服务架构的演进路线图

根据服务的个数和复杂度来分，大致可以分成如下4种架构：单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。

此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。

 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

 分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。

 弹性计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

本论文研究分布式服务架构和弹性计算架构,服务实例数在1000个以上的应用。

### 2.2 服务治理需求和分析

以下为服务治理的几点主要的需求和对应的分析：

在大规模服务化之前，应用可能只是通过RMI或Hessian等工具，简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过硬件进行负载均衡。

1.当服务量越来越多时，服务URL配置管理变得非常困难，硬件负载均衡器的单点压力也越来越大。此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。然后通过在消费方获取服务提供方地址列表，实现软负载均衡和Failover，降低对硬件负载均衡器的依赖，也能减少部分成本。

所以，框架中需要服务注册与发现和软负载均衡模块。

2.当进一步发展，服务间依赖关系变得错踪复杂，甚至可能分不清应用之间启动的先后顺序，架构师就不能完整的描述应用的架构关系。 这时需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。

所以，需要一个服务依赖关系和服务调用链跟踪系统。

3.服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要确定使用的机器数量，并且要确定扩容的时间点。

为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。

其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

所以，为了做服务容量评估，依次需要服务监控与统计，然后作服务使用情况报告，最后就可以把服务权重动态调整。

4.当规模继续扩大，应用之间不再是扁平的对应关系，而是分层的关系，比如核心数据层，业务集成层等，即使没有出现循环依赖，也不允许从低层向高层依赖，以免后续被逼循环依赖。

这时，需要在注册中心定义架构体系，列明有哪些层的定义，每个服务暴露或引用时，都必须声明自己应用属于哪一层，这样注册中心能更快的发现架构的腐化现象。

所以在服务分层架构中，也需要服务依赖关系的功能。

5.服务应用和Web应用是有区别的，它是一个后台Daemon程序，不需要Tomcat之类的Web容器。如果以Web应用为主，规范都是按Web应用的规范，就不得不把服务跑在一个不是必须要用的Web容器里。 因此，需要实现一个非Web的容器，只需简单的Main加载Spring配置即可。

所以，服务部署方式也是一个要考虑的功能点。

6.上下服务之间的SLA只是一个约定，确保服务运行时质量是关键问题。

比如，一个应用很重要，一个不那么重要，它们调用同一个服务，这个服务就应该向重要应用倾斜，而不是一视同仁，当支撑不住时，应限制不重要应用的访问，保障重要应用的可用。这时，就需要服务路由，控制不同应用访问不同机器，比如应用分离、读写分离。

所以，服务框架中需要服务路由的功能模块。

7．在实际生产中会遇到如下的场景，当某服务突然多了一个消费者，这个消费者的请求量直接把服务给拖跨了，其它消费者跟着一起故障。

首先，服务提供方需要流控，当流程超标时，能拒绝部分请求，进行自我保护。

其次，消费者上线前和提供者约定服务质量等级协定(SLA)，SLA包括消费者承诺每天调用量，请求数据量，提供方承诺响应时间，出错率等，将SLA记录在监控中心，定时与监控数据对比，超标则报警。

所以，服务流控是个容错特性，能提高服务的稳定性。

8.随着服务的不停升级，故障不可避免，每次核心服务一旦宕机，影响面很大，如何控制故障的影响面，服务是否可以功能降级或者资源劣化是需要考虑的问题。

应用间声明依赖强度，哪些功能强依赖，哪些弱依赖，然后基于依赖强度，计算出影响面，并定期测试复查，加强关键路径上的服务的优化和容错，清理不该在关键路径上的服务。

提供容错伪装(Mock)数据，伪装数据也应可以在注册中心在运行时动态下发，当某服务不可用时，用伪装数据代替，可以减少故障的发生，比如某验权服务，当验权服务全部宕机后，直接返回false表示没有权限，并打印错误日志报警。

所以，服务框架需要服务降级的功能，具体可以实现为服务伪装(Mock)容错。

9.有些情况下并不是所有服务的访问量都大，很多的服务都只有很小的访问量，却需要部署两台提供服务的机器，进行HA互备。如何减少浪费的机器，此时可能需要让服务容器支持在一台机器上部署多个应用，可以用多JVM隔离，也可以用ClassLoader隔离。隔离的机制也可以保护服务实例之间不相互影响。

所以，隔离是个容错特性，能提高服务的稳定性。

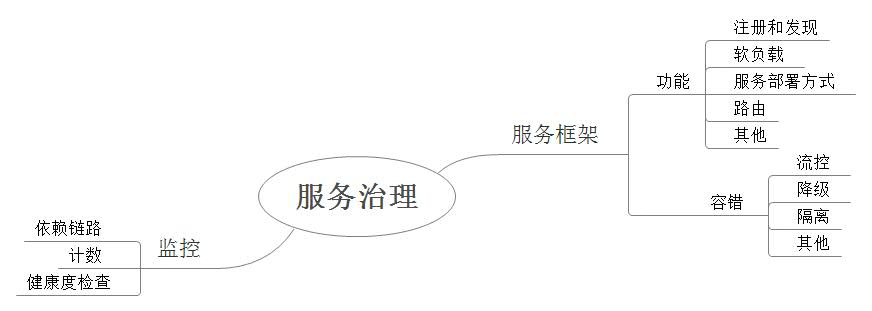
****

图2.2 服务治理需求

如图2.2所示为服务治理主要的需求，也是对以上这些需求的总结，后续的章节也会按照各个需求独立论述。服务治理以服务框架为主，服务监控为辅。监控服务后，可以对运行时的服务做出动态变更，比如权重的变更。也就是说服务监控后的结果可以反作用于运行时的服务，服务的运行时状态再在监控中展示，如此形成闭环。

如图2.3所示为服务治理的分层结构，服务框架在前端层时是服务消费者，服务框架在核心层时为服务提供者。服务治理外围的重要系统包括服务注册中心，监控中心和调度中心。服务注册中心主要存储服务提供者的元数据。监控中心监控服务运行时的状态。调度中心在容量评估后做服务的缩容或者扩容。

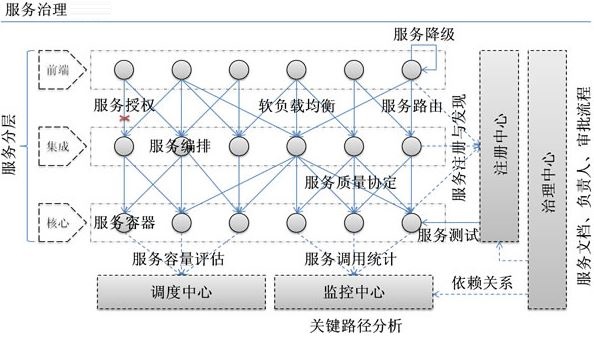


图2.2 服务治理的分层结构

### 2.3 服务框架中的角色

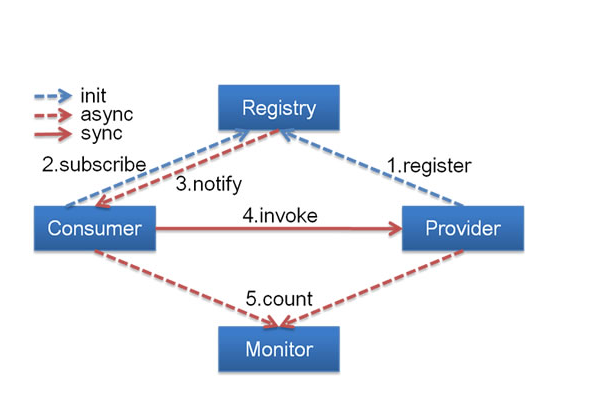


图2.4 分布式服务框架中的角色

如图2.4所示，图中可以分为以下几种角色：

Provider:提供服务方称之为“服务提供者”。

Consumer:调用远程服务方称之为“服务消费者”。

Registry:服务注册与发现的目录服务称之为“服务注册中心”。

Monitor:统计服务的调用次调和调用时间的日志服务称为“服务监控中心”。

服务框架有以下几种特性：

连通性

连通性描述了图2.4中角色之间的通信和交互关系，主要有如下几点：

1. 注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小

2. 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示

3. 服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销

4. 服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销

5. 注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外 。注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者

6. 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表

7. 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

健状性

健壮性描述了系统的可用性，整个系统的可用性是由服务提供者集群和注册中心集群提供保障的，不会出现单点故障的问题。各个角色的健壮性主要有如下几点：

1. 监控中心宕掉不影响使用，只是丢失部分采样数据。
2. 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务 。
3. 注册中心对等集群，任意一台宕掉后，将自动切换到另一台。
4. 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯。
5. 服务提供者无状态，任意一台宕掉后，不影响使用 。
6. 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复。

伸缩性

健壮性描述了系统的可扩展性，各个角色的伸缩性主要有如下的几点：

1. 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心

2. 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

### 2.4 服务框架技术选型

Hessian是老牌的二进制web服务协议，基于 HTTP协议进行传输。Hessian 通过其自定义的串行化机制将请求信息进行序列化，产生二进制流。但是0.1.x的hessian版本性能不高，瓶颈在io模型和线程模型上，吞吐量上不去。

Hessian是一个轻量级的remoting on http工具，使用简单的方法提供了RMI的功能。Hessian没有自身的线程模型，在web容器中实现的时候用的是容器的线程模型，比如在tomcat6.0中部署的时候，使用的是tomcat的基于BIO同步阻塞线程模型，客户端发起请求时要同步阻塞，等返回后才能执行后面的逻辑，所以吞吐性能上不去.

Hessian默认使用远程方法调用通信风格。服务之间的通信使用远程方法调用。客户端使用基于发送/请求的协议来发送请求到服务。这是常用简单的方式。系统没有中间的broker，所以比较简单。在交互的时间段里，客户端和服务端必须可用，这样减少了可用性。

Akka是JAVA虚拟机平台上构建高并发、分布式和有弹性的消息驱动应用的工具和运行时环境。Akka的Supervisor（监督者）的继承树有“let-it-crash”的语意，适合于稳定性要求高的系统，适合写容错自愈从不会宕机的应用。akka被设计成在分布式的环境里工作,所有actor之间的交互使用真正的异步消息来发送，所有的都是异步的。Akka本身实现了一个分布式轻量级的消息系统。但所有都是异步的也有问题，调试比较麻烦。

因为akka是全异步化和并行化的，相对于hessian吞吐和延迟上性能高了很多。Akka也是基于网络层TCP，相对于使用HTTP的hessian协议有更高的性能。

Akka使用基于消息的通信风格。服务间通信使用异步消息。通过在消息通道之间交换消息来做服务通信。这种方式是松耦合，解耦了客户端和服务端。消息broker缓存消息直到消费者有能力处理他们，这样增强了可用性。并且提供了大量的通信模式，其中包括请求/响应，通知，请求/异步响应，发布/订阅，发布/异步响应等。但是消息broker必须高可用，所以导致了broker附加的复杂性。

Hedwig 0.1.x版本选用了Hessian作为通信框架。 为了更高的性能提升，0.2.x选用了AKKA作为通信框架。Akka完全使用消息模式来实现，Actor和Actor之间用Mailbox队列来收发消息，某个actor失效重启后，能继续消费Mailbox，这样保证了不会丢消息，所以框架的可用性也大大提高了。

Akka有在ebay和亚马逊使用，国内用户相对少，据笔者了解在服务框架中没有用过。一号店hedwig使用akka构建服务框架是相对有特色和独树一帜的。

### 2.5 本章小节

本章首先讲述了服务化的背景，然后介绍了服务治理的需求，最后讲述了服务框架的架构中的角色和技术选型。

# 3 服务框架功能特性

分布式服务框架提供了高性能和透明化的RPC远程服务调用方案。

从整个分布式服务框架原理入手，抽取核心功能，服务框架能抽象出以下四层的架构。RPC层包括底层通讯框架，序列化和反序列化框架等。过滤器链（filter链）层提供多种切面供框架自身和使用者扩展，服务调用的责任链。例如负载均衡，通知机制，失败重试机制等。服务层包括客户端使用的Java动态代理机制，服务端使用的Java反射机制。业务层包括业务的服务抽象接口定义和实现类。服务由业务开发者来实现，平台负责将业务接口发布（export）为远程服务。

分布式服务框架特性主要包括功能特性和非功能性特性，功能特性如图3.1所示。下面就逐一分析各种特性，后续章节会介绍各个特性在hedwig里的设计。

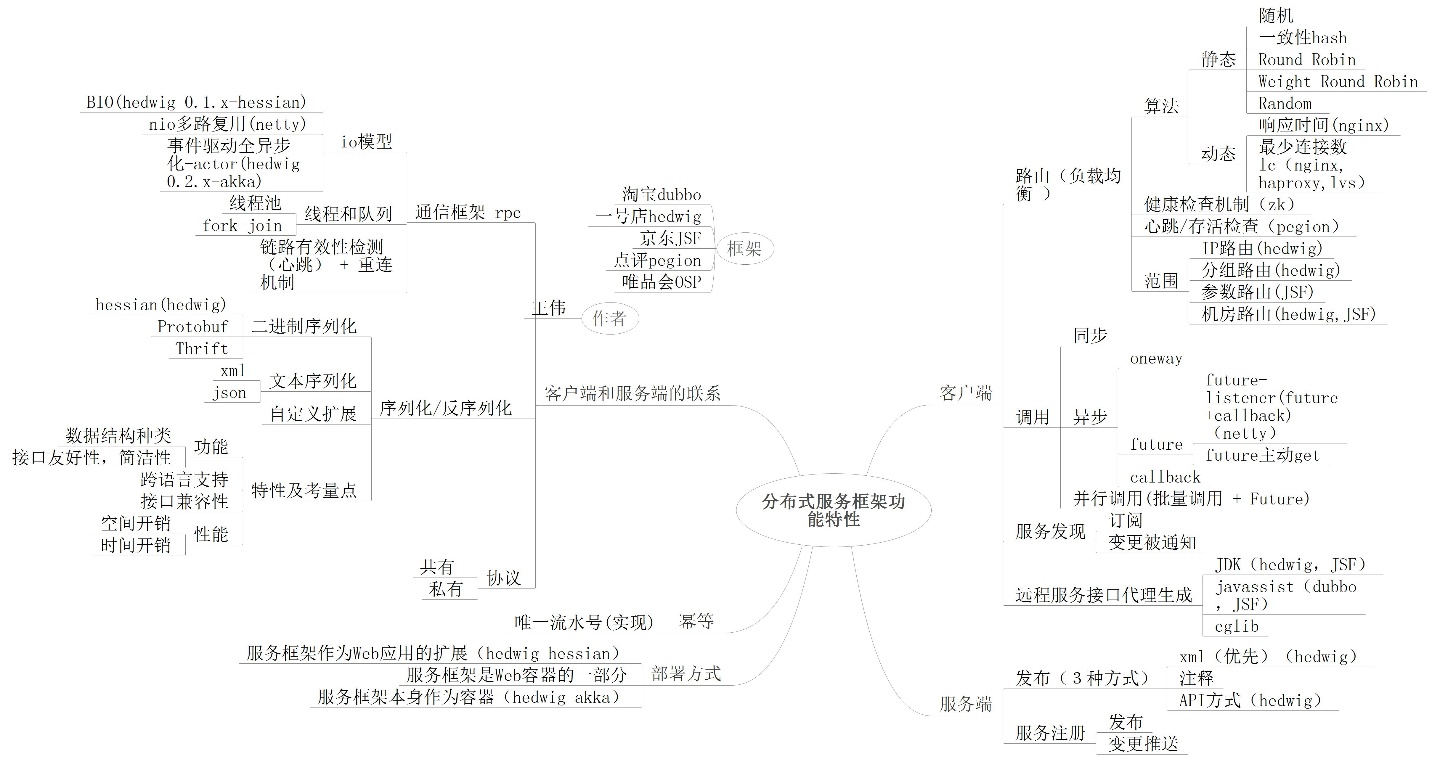


图3.1 分布式服务框架功能特性

### 3.1 服务注册和发现

对于服务提供者，它需要发布服务，由于应用系统的复杂性，服务的数量、类型不断膨胀；对于服务消费者，它最关心的是如何获取到它所需要的服务。

如何有效的管理服务订阅/发布，避免硬编码地址信息是分布式服务框架需要解决的一个问题。 通过将服务统一管理起来，可以有效的优化内部应用对服务发布/使用的流程和管理，服务注册中心就是专门用来管理服务订阅/发布的配置管理节点。

注册机制中的角色

注册机制中包含3种角色。服务提供方（provider）是发布服务并承载业务的服务提供者。服务消费方（consumer）是调用远程服务的消费者。服务注册中心（registry）是分布式服务框架的目录服务器，存储服务的元数据.

 关键功能特性设计

注册机制原理如图3.2所示：

1.服务端启动时，依据配置的服务发布信息向注册中心注册自身提供的服务。

2.消费者启动时，根据消费者配置的服务消费信息向注册中心订阅自己所需的服务，消费者在本地保存路由表。

3.注册中心返回服务端地址列表给客户端。如果有变更，注册中心主动推送变更给客户端，客户端刷新本地的路由表。

4.客户端从本地的服务端地址列表中，基于某种负载均衡算法选择一个服务端进行调用。

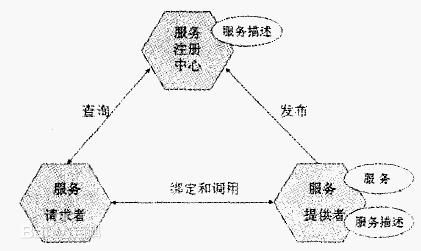


图3.2 服务注册原理

 订阅发布机制

1. 对变化的监听和主动推送能力

消费者可以监听一个或者多个服务目录，当目录名称或内容发生变更时，消费者可以实时的获得变更的数据或变更后的结果信息。

服务提供者可以发布一个或多个服务，动态修改服务名称或服务内容等，可以主动将修改后的数据或结果推送给所有监听此服务目录的消费者。

1. 订阅发布机制的优点

透明化路由是指服务端和客户端相互解耦，客户端不需要硬编码服务端地址。

服务健康状态检测是指如果服务端宕机，由服务注册中心实时通知客户端。

动态发现是指将新增的服务端地址信息推送给客户端，客户端刷新本地路由表后可以访问新的服务端。

### 3.2 服务路由和负载均衡

分布式服务框架上线运行时都是集群组网，这意味着集群中存在某个服务的多实例部署，消费者如何从服务列表中选择合适的服务提供者进行调用，这就涉及到服务路由。分布式服务框架要能够满足用户灵活的路由需求。负载均衡策略是服务路由的关键部分。分布式服务框架通常会提供多种负载均衡策略。如下是几种负载均衡算法：

静态负载均衡算法

1. 一致性hash:一样请求总是发到同一提供者。

基于虚拟节点，当某台提供者宕机，原来发往该提供者的请求平摊到其它提供者，不会引起网络震荡。

2. Round Robin:轮询算法

顺序循环访问服务的提供者的列表，到达上限后，重新归零，继续顺序循环。

轮询算法存在慢的提供者累积请求问题，比方:某台机器很慢，但没宕机。当请求调到这台时就夯住了,久而久之，所有请求都卡在调到这台机器上。

3. Weight Round Robin:轮询算法

基于权重的轮询算法,按公约后的权重设置轮循比率。

4. BROADCAST：广播算法

调用幂等的多个方法,如果其中有一个返回值，放弃调用其他的返回值。

 动态负载均衡算法

1. 服务调用时延算法

客户端缓存所有服务端的服务调用时延， 周期性的计算服务调用平均时延，

然后计算每个服务端服务调用时延与平均时延的差值，根据差值大小动态调整权重，保证服务时延大的服务端接收更少的消息，防止消息堆积。

通过动态自动权重调整消除服务调用时延的震荡范围，使所有服务端的服务调用时延接近平均值，实现负载均衡。比如Nginx（商业版）用的是最小平均响应时间算法。

1. 最少连接数算法

找到当前连接数最小的提供者，把新的连接请求分配到给它。

比如Nginx， haproxy，LVS都有最少连接数算法（lc）。变种算法有Dubbo LeastActive算法,最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。越慢的服务端的调用前后计数差会越大，所以可以使慢的提供者收到更少的请求。

服务路由的种类

1. 基于权重路由：如果机器配置低权重就低，如果机器配置高权重就高；

2. 基于IP路由：某个服务只能调用某几个机器；

3. 基于分组路由：自动根据配置调某个分组；

4. 基于参数路由：根据参数访问不同的节点或者根据方法名作读写分离；

5. 基于机房路由：同机房优先调用。

### 3.3 序列化和反序列化

服务提供者和消费者通过网络进行通信，对象需要进行序列化和反序列化，常见的序列化和反序列化方式很多，如何选择是重点也是难点。

 概念

序列化:将对象编码为字节数组，用于数据持久化，网路传输。

反序列化:把从网路磁盘等读取的字节数组还原成原始对象。

进行rpc调用时，需要使用特定的序列化技术，对网络传输的对象做编码或者解码。

 与通信框架的关系

序列化与通信框架不是强耦合的关系，通信框架提供的编解码框架可以非常方面的支持用户通过扩展实现自定义的序列化格式。用户也可以在应用程序及其他位置实现对象的序列化和反序列化，通信框架的编解码接口作为可选插件，并不强制用户一定要在通信框架内部实现消息的序列化和反序列化。

与通信协议的关系

序列化与通信协议是解耦的，同一种通信协议可能由多种序列化方式承载，同一种序列化方式也可以用在不同协议里。

比如，http协议可以用xml/json来承载消息体，也可以是图片附件等二进制流媒体协议。

 多种序列化方式

整体上看，序列化反序列化分为两种，文本和二进制。序列化反序列化框架要有以下特性：

1. 默认有多种常用的序列化反序列化方式，文本例如xml/json，二进制例如thrift和protocol buffer。

2. 框架有扩展性，可以扩展其他序列化方式。

 功能设计

在设计序列化/反序列化框架的时候，需要从功能，跨语言支持，兼容性，性能等多个角度进行综合考量，包括以下几点。

1. 功能包括支持的数据结构种类，接口是否友好，简洁。
2. 衡量序列化框架通用性的一个重要指标就是对多语言的支持。
3. 一个好的序列化框架应该支持数据结构的前向兼容性，例如新增字段，删除字段，调整字段顺序等。

### 3.4 服务调用

除了常用的同步服务调用之外，分布式服务框架还需要支持其他几种形式的服务调用。同步调用往往是默认的调用方式。异步相比于同步调用有两个优点：串行调用化为并行调用，提升调用效率，缩短业务线程阻塞的时间。同步化为异步,避免业务线程的阻塞，能减少服务的整体响应时间。

几个服务组合完成一个业务时，并行服务调用比串行服务调用更能减少响应时间。并行服务调用的目标有降低业务端到端的时延和提升整个系统吞吐量。

同步服务调用

分布式服务框架默认都需要支持同步服务调用，同步调用是常用的服务调用方式。

工作原理如下：客户端发起远程服务调用，用户线程做完消息序列化后，将消息投递到rpc框架，然后同步阻塞，等待通信线程发送请求并接收到应答后，唤醒同步等待的用户线程，用户线程获取到应答，然后返回。

异步服务调用

1. 单向调用方式。

单向调用是只管发送请求却不关心返回结果的方式。如果在NIO方式下使用，会比同步调用简单。

2. future方式

Future方式有两种：被动异步回调通知和主动获取，适用于不同的业务场景。在工程中会提供Future-Listener机制，例如Guava ListenableFuture实现异步调用。Future-Listener机制扩展了Future。

被动异步回调通知方式，如图3.3所示

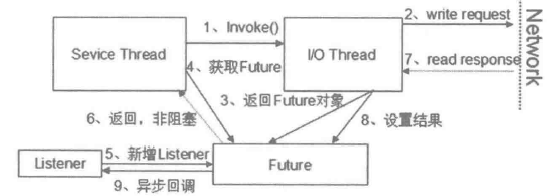


图3.3 future-listener被动调用机制

主动获取方式不添加监听器，如图3.4所示。 业务线程会阻塞，但相比同步服务调用，这种方式的阻塞时间更短。用户连续发起多次服务调用，然后顺序从rpc context中获取future对象，最终再主动获取结果。

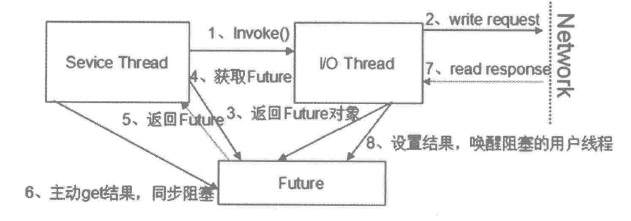


图3.4 异步服务调用主动get结果原理图

3.回调方式:请求方发送请求后会继续执行自己的操作，等到服务方有响应后进行一次回调。

因为回调本身的代码执行时间久等问题影响了io线程或者定时任务，所以还是建议用新的线程来执行回调。

单向调用是单方向的通知；回调是被动的，回调的执行不是在原请求线程中；future是主动控制超时，获取结果的方式，它的执行还是在原请求线程中。

 并行服务调用

可并行调用的条件有如下2点：一个长流程业务中调用几个个服务，并且对时延比较敏感，其中有些逻辑无上下文关联。多个服务没有相互依赖关系，没有严格要求的执行先后顺序。

并行服务调用的原理如图3.5所示：同时发起多个调用，流程fork后，再利用Future主动等待获取结果，进行结果汇集。实现并行服务调用的常用方案是批量串行服务调用。

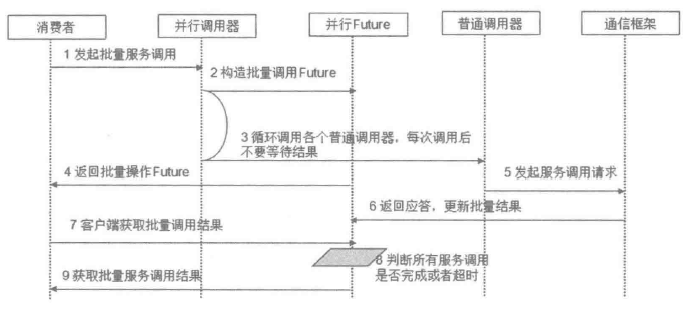


图3.5 批量服务调用原理图

### 3.5 RPC通信框架

单机版的本地方法调用变成远程服务调用之后，一个高性能的通用通信框架将成为分布式服务框架必不可少的组成部分。IO模型是通信框架性能是否好的关键点。可靠的底层通信框架才能保证服务框架的可靠性。

IO模型

1. BIO：同步阻塞模式的IO模型，其性能和可靠性有着巨大的瓶颈。服务端的线程数和客户端数是1比1的关系,所以BIO缺乏弹性伸缩能力。显而易见，要构建高性能，低延时，大并发的应用系统，同步阻塞I/O模型是无法满足性能线性增长和可靠性的。

2. NIO：I/O多路复用技术把多个I/O的阻塞复用到同一个reactor线程的阻塞上。系统开销小是I/O多路复用技术的优势。NIO的非阻塞模式可以开发高负载，高并发的网络应用开发。

一个reactor线程中的多路复用器可以同时轮询多个通道，由于JDK nio的底层使用了epoll代替了以前的select实现，所以它可以可以接入成千上万的客户端，因为它没有最大连接句柄的限制。

3. Actor：全异步化事件驱动。

通用并发编程模型中Actor模式是一种适用性很好的模型。Actor能提供很好的容错性，适合解决分布式的问题，也可以应用于共享内存架构和分布式内存架构。

Actor模型保留了可变状态，但不进行共享可变状态。Actor封装了状态，并通过消息与其他actor通信。Actor类似于面向对象中的对象。两者的区别是所有actor可以同时运行，而且，与OO式的“消息传递”不同，actor之间的消息传递是真实的在传递消息。

 可靠性设计

1. 链路有效性检测

周期性的对链路进行有效性的检测可以解决链路的可靠性问题。目前通用的实现方式是用心跳检测。心跳检测目的是确认对端存活并且能够正常收发消息。心跳检测机制有请求-响应型(Ping-pong形)和双向心跳型（Ping-ping型）。

发生心跳超时后要关闭链路，然后由客户端发起重新连接的操作，保证链路能恢复到正常的状态。

2. 重连机制

客户端检测到链路中断后，等待t时间，会发起重连操作，如果重连后还是失败，间隔周期t后再发起重连，直到重连成功。

在首次断连时客户端需要等待t时间后再重连，而不是失败后就马上重连，这是为了保证服务端有足够的时间释放资源。

任何情况下的重连失败，客户端都要保证自身的资源被及时释放，以防止内存泄漏的情况。

### 3.6 服务发布和引用

服务提供者需要支持通过配置、注释、API调用等方式，把本地接口发布成远程服务；对于消费者，可以通过对等的方式引用远程服务提供者，实现服务的发布和引用。服务发布和应用是指业务定义的抽象接口发布（export）为服务的过程。

 发布形式

服务发布通常存在三种服务发布形式,如表3.1所示：

表3.1 服务发布的三种方式

|  |  |  |
| --- | --- | --- |
| **服务发布方式** | **优势** | **劣势** |
| Xml配置化 | 1.服务框架对业务代码零侵入  2.扩展和修改方便  3.修改配置不需要重新编译代码 | 无 |
| 注释 | 1.对业务代码低侵入 2.扩展和修改方便 | 修改配置需要重新编译代码 |
| Api调用 | 无 | 1.对业务代码侵入较强 2.容易与某种具体的服务框架绑定 3.修改之后需要重新编译 |

采用xml配置化的方式发布服务是最佳选择，这也是大多数服务框架推荐采用配置化方式发布服务的主要原因。

服务提供者本地实现类封装成代理

对于服务端，将本地实现类封装成代理对象不是必需的；可以利用工具类将地址信息注册到服务注册中心。

采用动态代理的好处有3个：

1.服务的发布流程是相似的，通过抽象代理层，可以对服务发布行为本身进行封装和抽象。

2.通过动态代理对象，可以对服务发布进行动态拦截，方便平台和业务对服务发布进行个性化定制。

3.便于扩展和替换。

动态代理具体的技术实现可以有多种方式，例如JDK Proxy，Spring InvocationHandler，以及开源的字节码处理框架javassist，服务框架设计者可以根据具体场景需求选择相应技术。

 服务提供者信息注册

服务注册的目的有两个：供消费者订阅服务地址信息进行服务路由。基于服务注册中心的统一服务治理。

服务注册的结构有多种方式，例如按照主机地址，按照服务名或者URL，目录结构没有固定的结构，需要根据自己产品的实际需要进行判断。

服务注册要在服务发布的最后环节进行。一旦服务地址信息注册到服务注册中心，消费者就能够获得地址信息并发起请求调用；如果此时提供者还没有完成初始化，就会导致服务调用失败。

客户端本地调用转换成远程调用

原理是根据导入的服务端接口API和服务应用信息，生成远程服务的本地动态代理对象；它负责将本地的API调用转换成远程服务调用，然后将结果返回给调用者。

客户端远程服务调用

客户端将请求封装成协议消息，按照指定路由策略将请求发送到服务端。业务线程注册监听器回调或者同步等待。客户端反序列化响应消息，然后唤醒等待的业务线程。业务线程获得服务调用结果返回。如果客户端调用超时，则执行容错策略（重试策略），以此保证服务调用成功。

### 3.7 服务框架部署方式

在运行期服务框架与应用和容器的关系也称为服务框架的部署方式。在实际中服务框架自身的部署方式是比较重要的问题。

 方案1

如图3.6所示：分布式服务框架与应用一起打包,它作为应用的一个依赖包。这种情况下，服务框架变成了应用的一个三方或者二方库，并和应用一起启动。

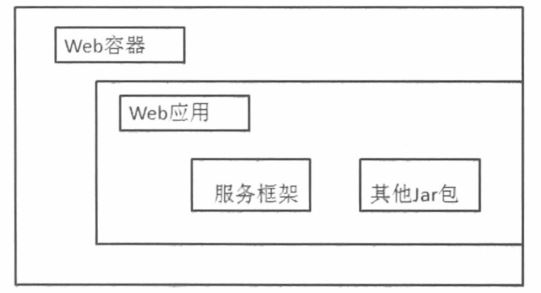


图3.6 服务框架作为Web应用的扩展

 方案2

如图3.7所示：分布式服务框架是容器的一部分， 这种方式比较少用，这里不再赘述。

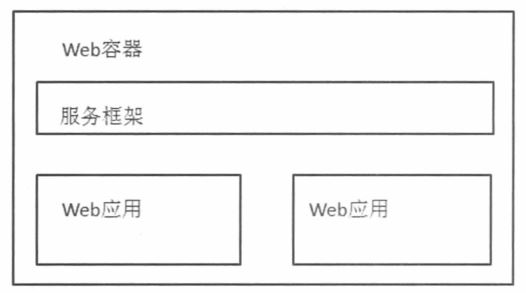


图3.7 服务框架是Web容器的一部分

方案3

如图3.8所示：有的情况下应用不需要容器（不是Web应用，或者不使用现有容器），那么，服务框架自身需要变为一个容器来承载远程服务。

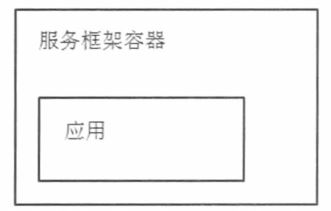
****

图3.8 服务框架本身作为容器

### 3.8 协议栈

不同服务在性能上适用不同协议进行传输。比如对接异构第三方服务时，通常会选择HTTP/Restful等共有协议；对于内部不同模块之间的服务调用，往往会选择性能较高的二进制私有协议。

 关键技术点分析

大部分服务框架都支持多协议，但是多协议却不是必须的。如果公司内部使用，则可以根据业务需要决定支持的具体协议，如果是开源的通用分布式服务框架，考虑到通用性往往默认会提供一些常用的协议，例如Web Servie和HTTP。多协议支持

是否支持多协议，需要根据业务的实际需要，公司对分布式服务框架的定位而定。如果确实有实际需要，支持多协议也未尝不可。

需要指出的是，服务框架需要具备通过扩展的方式支持多协议的能力，协议栈应该作为一个架构扩展点开放出来。

协议选择

对于Web Services，HTTP等公有协议缺少服务描述文件，服务订阅发布机制，不太适合作为SOA服务化的标准协议。

分布式服务框架主要是为了解决内部服务化之后业务接口跨进程通信问题，吞吐率，时延等性能指标是关键。在性能方面，私有协议往往可以根据业务的具体需求进行针对性优化，因此性能更优。

以web service公有协议为例，它的性能存在如下缺陷：

1. SOAP消息使用xml进行序列化，相比于PB等二进制序列化框架，性能低一大截。

2. SOAP通常使用HTTP协议，HTTP1.0/1.1不支持双向全双工通信，而且一般使用短连接通信，性能比较差。

如果没有特殊需求，分布式服务框架默认使用性能更高，扩展性更好的私有协议(二进制)进行通信。对于部分特殊需要与外部对接的服务，可以考虑引入HTTP/Restful等公有协议。

 功能设计

1. 功能描述

私有协议主要功能包括定义了私有协议的通信模型和消息定义，支持采用点对点长连接通信等功能。

2. 协议消息定义

通常协议栈的消息模型分为消息头和消息体。消息头存放协议公共字段和用户扩展字段，消息体则用于承载消息内容。以HTTP协议为例，请求消息头允许客户端向服务端传递请求的附加信息以及客户端自身的信息，常用的消息头关键字有Accept，Host等。

私有协议的消息模型与公有协议类似，也包含消息头和消息体。消息头通常包含服务调用相关的公共参数，消息体是个byte数组。

### 3.9 幂等

随着分布式系统及微服务的普及，因为网络原因而导致调用系统未能获取到确切的结果从而导致重试，这就需要被调用服务具有幂等性。多次执行幂等操作所产生的影响，均与执行一次的结果相同。如果想要从而从错误中恢复，重新处理消息时，幂等会很有用。在多次写操作，如果操作是幂等的，这样的操作也是安全的。

举个例子，如图3.9所示, 某个订单购物300元，以下图所示的负载发起一个调用。

<order>

<amount>300</amount>

<account>user1</account>

</order>

图3.9 订单购物300元

如果多次收到这个调用，会多次扣除用户300元。因此，这个调用不是幂等的。然而如图3.10所示可以让支付操作的调用变成幂等操作。

<order>

<amount>300</amount>

<account>user1</account>

<serialNumber>10001</serialNumber>

</order>

图3.10 订单购物300元

现在知道这个订单和一个特定的流水号10001相关。假定一个流水号只能支付扣减一次，我们可以在不减少总费用的情况下，再次应用这个订单。

所以，幂等通用的做法是生成唯一的一个序列号，以此来保证幂等性。

### 3.10 本章小节

本章讲述了分布式服务框架各个功能模块实现的原理。

# 4 服务框架功能组件和模式

系统往往由一些粗粒度的组件组合在一起，分布式服务框架也是由几个大的组件组成的。组件化的好处是组件之间是松耦合的，组件替换起来方便, 有可扩展性。 hedwig扩展底层Hessian RPC协议比较方便，如果要扩展新的gRPC也就是写一个Client Request Handler和Server Request Handler，其他的部分不需要变更。

本章会讲解每个组件的职责和抽象出来的模式。服务框架组件模式主要由以下几个过程实现，如图4.1所示。Client proxy、Requestor、Client Request Handler是客户端的组件，Server Request Handler，Invoker，Remote Object是服务端的组件，接口描述（Interface Description）和远程错误(Remote Error)是客户端和服务端的公用组件。由Client proxy构建请求发送到Requestor里，Requestor发送请求到Client Request Handler，Client Request Handler远程和Server Request Handler通信，并发送请求。Server Request Handler分配请求给Invoker，Invoker分配调用给Remote Object。Client proxy和Remote Object之间使用统一的接口描述。

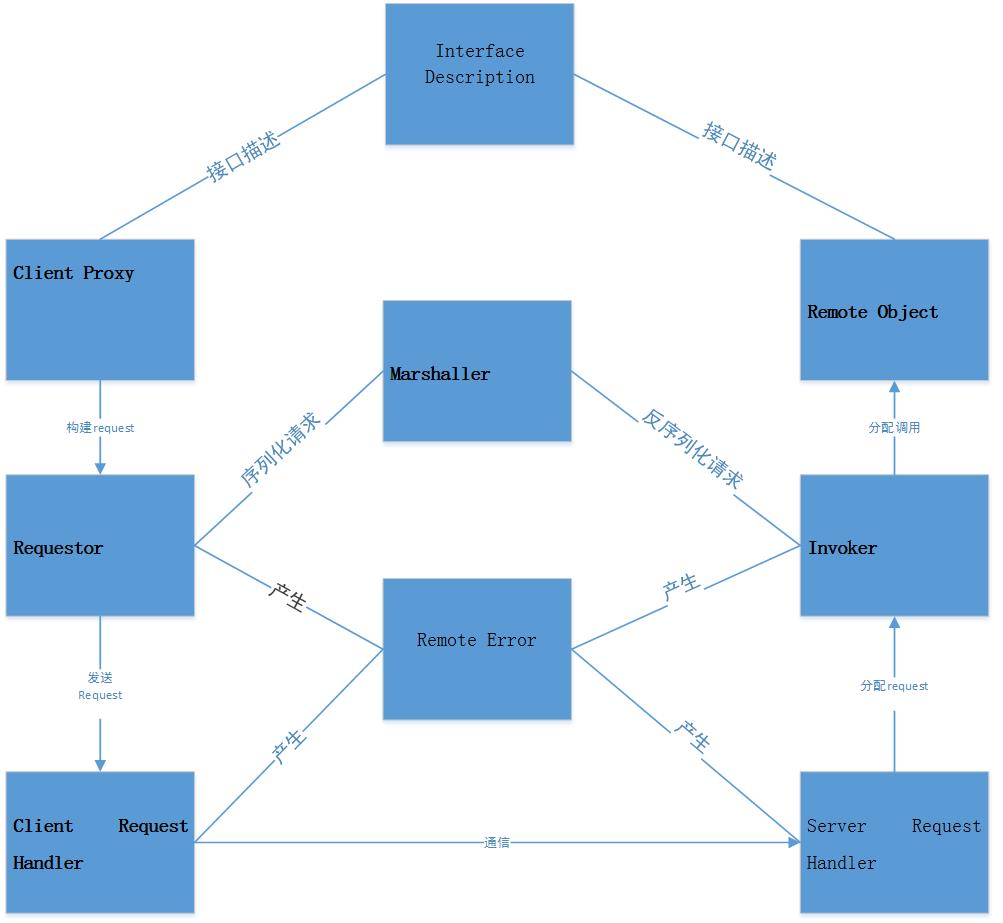


图4.1 服务框架组件模式

### 4.1 客户端代理

为客户端提供了一个本地接口，用于与远程组件进行交互。客户端能够以一种与位置无关的方式访问远程组件，使得远程组件看起来是与本地组件在一起的。

在客户端中提供一个客户端代理(client proxy)，它可以代替远程组件。Proxy提供了与远程组件相同的接口，并将客户端的调用映射到特定的消息格式和协议，以便跨网络发送这些调用请求。

Client proxy支持远程方法调用风格的IPC。这样调用本地组件和调用远程之间就没有api上的区别了，这提高了分布式应用内部通信的位置无关性。另外，client proxy可以为客户端屏蔽掉远程组件接口上的变化，这样可以避免在更新组件时产生涟漪效应。

### 4.2 请求者

封装了客户端远程通信的各种细节，比如封送，通过网络发送请求，允许客户以位置无关的方式访问远程组件。请求者(Requestor)负责创建和处理请求消息，并将请求发送给远程组件。

Requestor可以将它的一些子任务委托给其他的组件。Marshaler可以将一个具体的服务请求序列化为请求Message，以及将包含了结果的消息反序列化为一个具体的响应。

一些应用需要Requestor完成额外的工作，比如添加安全令牌，Interceptor可以通过提供统一的接口来封装这项任务。

Absolute Object Reference可以封装远程目标组件的位置和标识信息。与前面类似，当过程中出现了失败，无法由Requestor透明的处理时，将会返回给客户端一个Remoting Error.

### 4.3 客户端请求处理器

封装了客户端里底层进程间通信的细节，这些细节在统一的接口背后。职责包括创建连接，发送请求和分配结果，处理超时和错误。另外，客户端请求处理器(client request handler) 还要负责网络和计算资源的高效管理和利用，比如网络连接，内存和线程。

client request handler实现了Acceptor-Connector模式中的Connector角色，它可以使网络连接创建策略的变化独立于client request handler的其他职责。

由Connector创建的连接可以封装在连接处理程序中，后者扮演了Acceptor-Connector中Service Handler的角色。这种设计将连接视为一阶实体，这样可以支持对特定于连接的状态进行高效维护，同时可以处理产生于连接过程中的Remoting Error. 同时也可以支持可伸缩性，因为每一个连接处理程序都要运行在它自己的线程中，所以能同时处理来自多个客户端的请求，并同时回复多个客户端。

连接处理程序可以实现同步或者异步的通信策略。同步通信能够简化客户端的编程模型，但也会降低性能和吞吐量，而异步通信恰恰想法。有4种模式来实现异步通信模型-Fire and Forget， Sync with Server，Poll Object和Result Callback. 这4种模式为应对下面3个问题给出了4种不同的解决方案，这3个问题是：结果是否需要发送给客户端； 客户端是否需要接收确认信息；向客户端发出结果后，客户端是自主获取这个结果，还是需要回调函数的通知。

Connector和Client Request Handler的连接处理所使用的ipc机制，可以用Protocol Plug-in或者一系列Wrapper Façade来进行封装. 这两个模式都可以在统一的平台无关的接口之后隐藏ipc机制的细节。Protocol Plug-in允许运行时（重）配置ipc机制，但是会导致一些运行时的开销。相反，Wrapper Façade可以避免运行时的开销，但是只支持编译时的配置。

调用的结果，以及无法由client Request Handler处理的Remoting Errors,都会返回给发送相应请求的客户端。

### 4.4 调用者

可以在服务器组件收到来自远程客户端的请求时，替服务器组件屏蔽掉与网络相关的问题，比如请求的接收，解封送，分派请求。

调用者(Invoker)可以将多个子活动的职责委托给其他组件。Marshaler可以将消息Message反序列化为一个具体的服务，再将相应的服务结果序列化到结果Messages中。

Absolute Object Reference封装了特定组件实现的标识信息。当过程中出现了失败，无法由Invoke透明的处理时，将会返回给发送请求的客户端一个Remoting Error.如果Invoker无法找到组件的实现（比如，组件已被重新部署或者组件在高负载的情况下暂时不可用）， 它可以把调用委托给Location

Forwarder.

### 4.5 服务端请求处理器

封装了服务端底层进程间通信的细节，这些细节在统一的接口背后。职责包括创建连接，接收和分派请求，发送执行结果和处理错误。另外，服务端请求处理器(server request handler) 还要负责网络和计算资源的高效管理和利用，比如网络连接，内存和线程。

server request handler需要一个事件处理基础设施，它会监听网络中到来的连接请求，建立请求的连接，将服务请求分派到合适的应用组件的方法上。这个基础设施必须可以同时接收和处理多个连接和服务请求，这样才能得到适当的延迟时间，吞吐量和可伸缩性。它的核心可以实现为Reactor或者Proactor，这取决于处理接收事件的机制是串行的，还是由中断驱动的。

### 4.6 接口描述

当客户端要使用client proxy调用远程对象的操作时，client proxy和远程对象的接口要一致对齐，序列化和反序列化要一致对齐，Client开发者需要知道远程对象接口。

因此，需要使用接口描述（Interface description）来描述远程对象的接口。接口描述作为Client proxy和invoker之间的契约。Client proxy和invoker之间要么使用代码生成要么使用运行时配置技术来遵守这个契约。

### 4.7 远程错误

远程通信往往是不可靠的。也是是说，分布式调用不可能是完全透明的。

除了远程对象自身的错误， 新的类型错误也会产生，比如网络失败，服务器宕机，或者是不可用的对象。

因此，为了客户端可以处理这种异常，需要定义分布式相关的异常和应用逻辑相关的异常,即远程错误(Remote Error)。Invoker, requestor, and request handlers侦测到异常，然后传播异常。

以上这些设计模式能清晰划分框架的组件边界，对设计出可插拔，可扩展的框架有借鉴意义。

### 4.8 本章小节

本章讲述了分布式服务框架各个功能组件实现的原理和模式。

# 5 服务框架容错原理和模式

### 5.1 集群容错

集群中服务调用失败后，服务框架需要能够在底层自动容错，上层应用对下层错误是无感知的，这样就提高了框架的可用性。容错策略很多，分别适用于不同场景。

 容错策略

**1.** Fail over

Fail over是指当发生远程调用异常时，重新选择查找路由到下一个可用的节点。

Fail over策略的设计思路如下:消费者路由操作完成后， 获得目标地址，调用通信框架的消息发送接口发送请求，监听服务端应答。 如果返回的结果是rpc调用异常（超时，流控，解码失败等系统异常），根据消费者集群容错的策略进行容错路由，如果是failover，则重新返回到路由handler的入口，从路由节点继续执行。 重新路由选择后，对目标地址进行比对， 防止重新路由到故障服务节点，过滤掉上次的故障服务端之后，调用通信框架的消息发送接口发送请求消息。

**2.** Fail cache

Fail cache是失败自动恢复的一种策略。

对时间延迟要求不是很高的服务、必须发送到指定的服务端、通知类型的服务的这样3种场景适合Fail cache机制。

服务调用失败通常是链路暂时不可用，服务流控，fullgc导致服务进程暂时夯住。当遇到以上这种情况，服务框架将消息缓存，等待周期t后，重新发送，直到服务端正常处理该消息为止。

Failcache中缓存对象要限制上限来防止内存溢出。Failcache中要限制定时重试的周期t，重试的最大次数等，并支持用户指定。重试达到最大上限任失败，需要丢弃消息，记录异常日志。

**3.** Fail fast

快速失败是指业务高峰期如大促，对于非核心的服务，只想调用一次，失败后不再次调用，为核心服务节约了宝贵的资源。

快速失败的设计很简单，拿到调用异常后，忽略异常，只记录异常日志。

### 5.2 服务降级

当访问量激增，非核心服务影响到核心流程的性能或服务出现大量问题时，仍然需要保证服务还是可用的，即使是有损服务。这时，可以根据一些关键指标进行自动降级，或者配置开关作人工降级。

人工开关降级

流量高峰时如果通过监控发现某些服务存在问题，这时需要将这些服务暂时关闭。如果服务之间存在激烈竞争，导致核心服务不稳定，可能会影响系统的用户体验。弱依赖的服务在大流量下停止调用.这时要对非核心服务做强制降级，直接返回空值，不发起远程服务调用,异常或者执行特定的本地逻辑，减少自身对公共资源的消费，把资源释放出来供核心服务使用。

有时发现突然调用量太大，可能需要同步转换为异步调用，此时可以使用开关来完成降级。另外，有时候不太明确该服务的逻辑是否正确，但又想让新开发的服务在线上作灰度测试，这时，可以设置降级开关，当新服务有问题时可以通过开关切换回老的正常的服务。还有多机房的情况，如果某个机房不可用了，则需要将这个机房的服务整体切到另一个机房，此时，可以通过开关完成切换。功能降级时也可以设置开关，暂时屏蔽掉某些功能。

人工降级操作是可逆的，当系统压力恢复正常水平，可以对已经人工降级的服务恢复正常。恢复之后，客户端重新调用远程的服务端，同时服务状态被修改为正常状态。

开关可以存放到配置文件，数据库或者Zookeeper/etcd. 我们需要通过配置方式来动态开启/关闭降级开关，在应用时，首先要封装一套应用层api方便业务逻辑使用。对于开关数据的存储，如果涉及的服务器/系统较少，则初期可以考虑使用配置文件进行配置。如果涉及的服务器/系统较多，则应该使用配置中心进行配置。实现时要做到不需要修改代码，不需要重启应用即可动态配置开关。

使用统一配置中心，目的是集中管理配置开关。不管是采取拉取或者推送模式的配置，都要考虑到并发连接数可能带来的问题。

系统自动降级

系统自动降级也称为容错降级.容错降级可用于业务放通，也常用于服务端在客户端执行容错逻辑，容错逻辑主要有rpc异常和service异常。rpc异常一般指超时异常，消息编解码异常，流控异常等。service异常指一般的业务异常。

系统自动降级大致可以分为以下4类：

1. 自动开关降级

自动降级是根据系统负载等资源使用情况进行降级。

1. 超时降级

当访问的远程服务调用响应慢，且该服务是非核心服务的话，在超时后可以自动降级。

在调用其他团队的远程服务时，可以和对方约定一个最大服务响应时间，如果超过了这个时间，则自动降级。在实际应用中，超时时间和超时重试次数一定要配置好。

3. 统计失败次数降级

当依赖某些别的公司提供的外部服务时，可能会碰到不稳定的api，当失败调用次数到达某个阀值时会自动降级（熔断器）。然后，通过异步线程去检测服务是否已经恢复，如果恢复则取消降级。

4. 限流降级

当用户抢购某些限购产品时，可能会因为访问量剧增而导致后端系统崩溃，如果超过限流阀值，后续请求会被降级。

### 5.3 服务可靠性

相对于传统的本地调用，跨进程的分布式服务调用面临的故障风险更高。对于应用而言，分布式服务框架需要具备足够的健壮性，在框架底层能够拦截并向上屏蔽故障，业务只需要配置容错策略，即可实现高可靠性。

服务状态监测

在分布式服务调用时，某个服务提供者可能已经宕机，如果采用随机路由策略，消息会继续发送给已经宕机的服务提供者，导致消息发送失败。为了保证路由的正确性，消费者需要能够实时获取服务提供者的状态，当某个服务提供者不可用时，将它从缓存的路由表中删除掉，不再向其发送消息，直到对方恢复正常。

1. 基于服务注册中心状态监测

工程中，通常会使用注册中心对服务端进行状态监测，当监测到服务端不可用时，会将故障状态广播到所有的客户端，客户端被通知到后，根据故障信息把故障节点排除在路由表之外。

客户端监听订阅的服务端的Node列表，当Zookeeper服务端检测到Zookeeper客户端会话超时之后，就会将该会话对应的Node删除，并将Node删除事件通知到所有监听该Node的消费者，客户端根据通知消息中的服务端信息更新路由表，不再向故障节点发送消息。

2. 链路有效性检测机制

服务消费者和提供者之间默认往往采用长链接，并且通过双向心跳监测保障链路的可靠性。

在某些特殊场景中，服务端和注册中心之间，客户端和注册中心之间网络可达，但是服务端和客户端之间网络不可达，或者提供者和消费者之间链路已经断连。此时，如果消费者任旧向链路中断的提供者发送消息，写操作将会失败。注册中心并不能检测到服务端异常。

因此，一般要使用注册中心检测加上服务端和客户端之间的链路有效性检测来保障系统的可靠性。

当客户端通过双向心跳检测发现链路故障之后，会主动释放链路，并将对应的服务端从路由表中删除。当链路恢复之后，重新将恢复的故障服务端地址信息加入地址列表中。

服务健康检查

在集群组网环境下，由于硬件性能差异、各服务提供者的负载不均等原因，如果采用随机路由分发策略，会导致负载较重的服务提供者不堪重负被压垮。

使用健康度监测机制，可以对集群的服务实例进行检测，根据检测结果对健康度打分，得分较低的亚健康服务节点，路由权重会被自动调低，发送到对应节点的消息相比于其他健康节点会少很多，这样就能够实现“能者多劳，按需分配”，实现更合理的资源分配和路由调度。

服务的健康度检测通常需要采集如下性能指标：

1.服务调用延迟,QPS,调用成功率

2.基础资源使用情况，例如堆内存，CPU使用率等

健康度评分通常由独立的监控节点负责，服务端将性能数据周期性汇报到监控节点，性能汇总和监控节点根据内置的健康度算法对各个服务端做健康度打分，然后周期性的将服务端健康度信息下发给客户端，客户端根据健康度评分动态调整路由权重，实现按需路由，向处理能力强的节点发送更多的消息，保护亚健康服务节点。

### 5.4 服务隔离

隔离是指将系统或资源分割开，系统隔离是为了在系统发生故障时，能限定传播范围和影响范围，即发生故障后不会出现滚雪球效应，从而保证只有出问题的服务不可用，其他服务还是可用的。资源隔离通过隔离来减少资源竞争，保障服务间的互相不影响和可用性。

 服务故障隔离

1. 线程隔离

线程隔离指线程池隔离。把分类过的请求交给不同的线程池处理。当某业务的请求处理线程膨胀消耗过多的资源时，不会将故障扩散到其他线程池，从而确保其他服务可用。

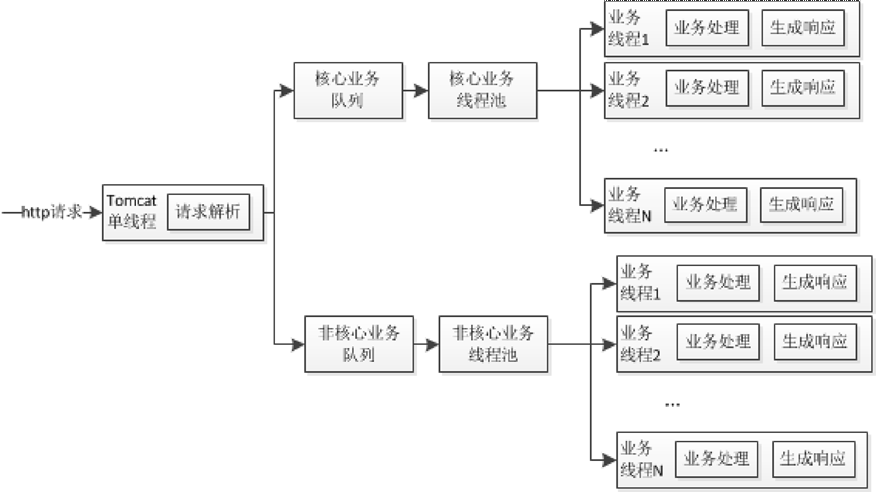


图5.1 tomcat中的线程隔离

如图5.1所示，Tomcat中可以配置多个线程池，核心业务分配核心业务线程池，线程池可以大一点，优先级高一点；非核心业务分配非核心业务线程池，线程池相对小一点，优先级低一点。

使用tomcat时，多个服务会共享一个线程池，默认线程池大小是150。假如其中一个服务访问的DB响应很慢，这会使得服务时延增加，多数线程会阻塞等待响应返回，导致整个web服务器线程池都被该服务占用。

线程或者信号量隔离可以来解决这个问题。使得某个服务的线程池满了也不会对其他服务造成严重的故障。

线程隔离包括三种粒度的隔离实现, 如下所示

粗粒度实现:服务分组加上线程池，每个服务分组配置一个隔离线程池，相同分组的情况配置共用一个线程池。

细粒度实现:服务和线程池，服务分组中每个服务配置一个隔离线程池。

混合实现:对核心服务单独设置线程池，一个服务分组配置一个线程池。

2. VM故障隔离

通过将基础设施层虚拟化,将应用部署到不同的vm中，利用vm对资源层的隔离，可以实现更高层次的服务故障隔离。

将核心服务和非核心服务分别部署到不同的vm中，利用vm的cpu调度，网络I/O,磁盘和内存等资源限制，实现物理资源层的隔离。

3. 物理机故障隔离

每个服务实例部署在一个物理机器上。

4. 集群隔离

当某个服务不稳定并可能影响到其他的分组时，要考虑为服务提供独立的服务集群(服务分组)。

5. 机房故障隔离

多机房部署是有一定规模互联网公司的标配，每个机房的服务都配置了自己的服务分组。本机房的服务应该只调用本机房服务，除非本机房服务都宕机，不会进行跨机房调用。

机房服务发生故障也是有可能的，这时可以通过负载均衡器将请求全量切换到另外一个机房，或者考虑服务能否自动重试其他机房的服务，从而可以提升系统可用性。

在服务分组名中加上机房名是一种可用的办法。

6. 基于接口,方法,参数的路由隔离

当系统运行时，如果并发请求量比较大，可能所有工作线程都已经在工作了，如果这时有新的请求进来就需要排队，当然，这是非常正常的逻辑。

但是如果这个服务端的某个方法是一个很慢的方法，线程很快就都陷入了执行这个方法的状态，之后再进来的请求就都需要排队等待，而且等待的时间是非常长的。

有两种应对这种情况的方案：

第一种方式：增加可执行的线程总数来保证在实际运行的过程中总是有可用线程提供服务，但这样不太经济也不太可控。实际中计算需要的线程总数是很困难的事情

第二种方式：隔离这些资源使得快慢不同，重要级别不同的方法之间互不影响。让其中对于某些服务的请求到一部分机器，让另一些服务的请求到另一部分机器。

从客户端的角度来讲，控制同一个集群中不同服务的路由是一种可行方案。可以实现为基于接口，方法，参数的路由。

 舱壁隔离模式

舱壁(Bulkhead Isolation)是一种工程中很通用的模式，可以选择隔离服务器的CPU，集群中的一个服务器，或者是应用中的线程池。

### 5.5 流量控制

限流的目的是通过对并发请求或者一个时间窗口内的请求进行限速来保证系统的稳定性。如果达到限制速率就可以降级，排队或等待，拒绝服务。在压测时能找出每个系统的处理峰值，然后通过设定峰值阀值，来防止当系统过载时，通过拒绝处理过载的请求来保障系统可用。另外，也应根据系统的吞吐量，响应时间，可用率来动态调整限流阀值。

当资源成为瓶颈时，服务框架需要对消费者或者提供者做限流，启动流控保护机制。流量控制有多种策略，比较常用的有针对访问速率的静态流控、针对资源占用的动态流控、针对消费者并发连接数的连接控制和针对并行访问数的并发控制等。静态流控和动态流控是对系统的整体流控。

 静态流控

静态流控是针对消费端访问速率进行控制，一般会根据约定的TPS做全局流量控制。

1. 传统静态流控设计方案

静态流控设计采用预分配方案，根据静态流控阀值和集群服务节点个数，计算每个服务节点分摊的tps阀值。系统运行时，每个服务按照自己的阀值进行流控， 如果超出阀值就拒绝请求访问。

服务框架启动时，将节点的静态流控阀值加载到内存中，服务框架通过handler拦截器在服务调用前做拦截计数，当计数器在指定周期t达到qps上限时，启动流控，拒绝新的请求消息接入。

要注意的是，到达流控阀值之后拒绝新的请求消息接入，不能拒绝后续的应答消息，否则会导致客户端超时或者触发failover，增加服务端的负载。

分布式服务框架的一个特性是服务的动态上下现和自动发现机制，在运行期间服务节点数会随业务量的变化而变化，在这种场景下静态分配方案显然无法满足需求。

**2. 动态配额申请制**

动态配额申请制的工作原理如下：

系统部署时，根据服务节点数和静态流控qps阀值，预先分配一定比例的配额做为初始分配，剩余的额度放在配额资源池。如果服务使用完了配额，会主动向注册中心申请。如果总配额被使用完，各个申请配额的服务节点只能拿到零配额，服务节点对后续的请求进行流量控制。

动态配额申请制的优点如下：

1).实时性高:负载情况在各个节点内计算，性能kpi数据在本地内存中计算获取。

2).流控精准性:各个服务节点根据自身负载情况去申请配额，保证性能高的节点有更高的额度，性能差的自然配额就少，这样能够更合理地调配资源。

实践验证表明，采用动态配额申请制的静态流控更精确，在实战中效果也更好。

 动态流控

动态流控不是对流量或者访问速度做精确控制，而是拒绝过大的负载，目标是为了保证系统的正常运行。

动态流控分为多个级别，每个级别被拒绝掉的消息比例不同，流控系数也就不同。每个级别都有自身的流控阀值，一般会支持在线动态调整。

 服务接口限流

1. 限制并发量和限制连接数

限制并发量是对线程并发数进行控制。并发控制有针对服务端的全局控制和针对客户端的局部控制两种形式。限制连接数是为了防止客户端连接数过多导致服务端压力过大，需要针对连接数进行流控。

2. 限流总并发/请求数

如果接口会有突发流量，又担心负载太大造成系统崩溃，这个时候需要限制接口的总并发/请求数。这种方式适合对需要过载保护或者可降级业务的服务进行限流。

一些开放平台会限制用户调用某个接口的请求量，这样就可以用计数器来实现。这种方式没有作平滑处理，也是简单的限流，具体根据实际情况按需实现。

3. 限流时间窗内请求数

一个时间窗口内的请求数，比如一些基础服务会被很多其他上层应用调用，要对每秒/每分钟的调用量进行限速。

4. 根据来源做流控

对于同样的接口或方法，根据不同来源设置不同的限制。 一般在基础服务上，也就是多个集群使用同样的服务时,根据请求来源的不同级别进行不同的流控处理。

 限流算法和服务接口突发请求整形

限流算法有计数器算法，漏桶算法，令牌桶算法。

令牌桶算法，如图5.2所示：往桶中添加令牌有固定速率往，如果桶里有令牌则请求能被处理，当令牌数减到0时，就拒绝新的请求。

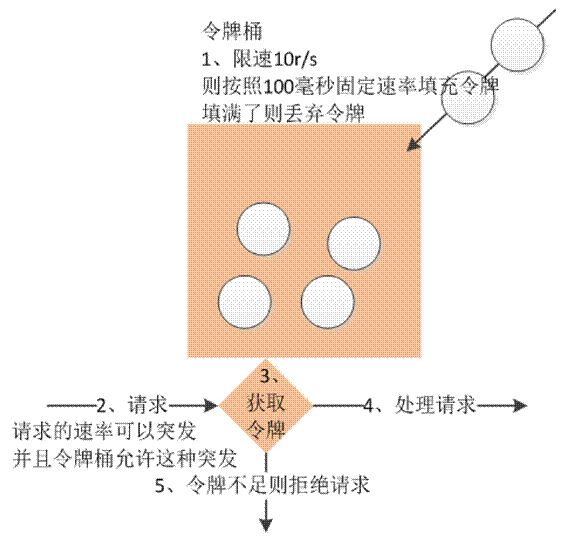


图5.2 令牌桶算法

漏桶算法，如图5.3所示：请求流入速率任意，请求流出有固定速率。流入请求数超过漏桶容量时，后续请求被拒绝。

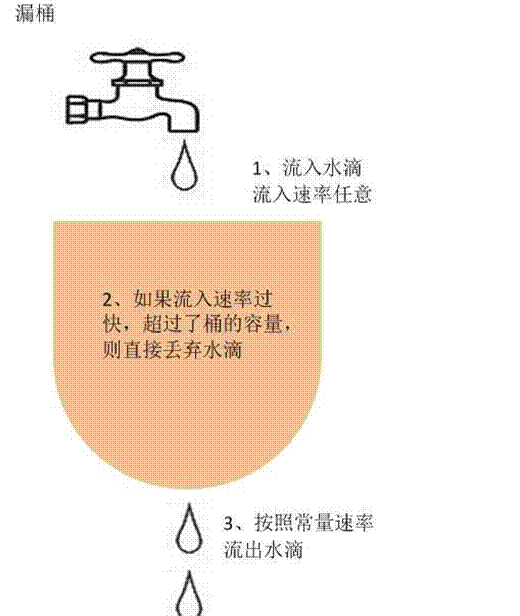


图5.3 漏桶算法

令牌桶限制的是平均流入速率，并允许一定程度的突发流量。漏桶限制的是常量流出速率，从而平滑突发流入速率。漏桶算法目的是平滑流入速率，令牌桶算法允许一定程度的突发流量。

一些场景中需要把突发请求整形为平均速率请求。令牌桶算法和漏桶算法可以满足这个场景。Guava RateLimiter提供的令牌桶算法可用于实现平滑预热限流和平滑突发限流。

 限流模式

限流(Rate Limiting/Load Shedder)也可以叫流量控制，主要有两种类型，本节提到的限流都是节流：

节流(Throttling):最容易想到的机制，就是丢弃流量。服务消费的很吃力，就处理一部分，丢弃余下流量。至于处理哪些和丢弃哪些，有throttleFirst、throttleLast和throttleWithTimeout策略来决定处理和丢弃哪些流量。

反压(Backpressure): 后端对前端流量的反向控制。消费者需要多少流量生产者就生产多少流量。

工程实现中Akka ,Storm,Spark Streaming,RxJava都实现了反压机制。

### 5.6 超时与重试机制

在实际开发过程中，太多故障是因为没有设置超时或者设置的不对而造成的。

如果应用不设置超时，则可能会导致请求响应慢，慢请求累积导致连锁反应，甚至造成应用雪崩。而有些中间件或者框架在超时后会进行重试（如设置超时重试两次），读服务天然适合重试，但写服务大多不能重试（如写订单，如果写服务是幂等的，则重试是允许的），重试次数太多会导致多倍请求流量，即模拟了DDos攻击，后果可能是灾难，因此，务必设置合理的重试机制，并且应该和熔断，快速失败机制配合。

在各个层面的系统都有超时机制，这里论述服务框架相关的超时。其中最重要的超时设置是网络连接/读/写的超时时间设置。

 中间件客户端超时与重试

中间件客户端超时与重试(如Dubbo，JSF，HttpClient等)需要设置客户的网络连接/读/写超时时间与失败重试机制。

提供端/消费端与注册中心之间进行服务注册/发现时可以配置timeout（调用注册中心超时时间，默认为5s）。提供端可以配置timeout（服务器端调用超时时间，默认为5s）。消费端可以配置timeout（调用端调用超时时间，默认为5s）。

 业务超时

业务超时也可以认为是服务端超时,如订单取消任务，超时活动关闭，还有如通过Future.get(timeout, unit)限制某个接口的超时时间。

业务超时分任务型和服务调用型两类。任务型超时比如订单超时未支付取消超时，活动自动关闭等，这属于任务型超时。服务调用型比如某个服务的全局超时时间是500ms，有多处服务调用，每处服务调用的超时时间可能不一样，此时，可以简单的使用future来解决问题，通过如future.get(3000,TimeUnit.MILLISECONDS)来设置超时。

 最佳实践

超时之后应该有相应的策略来处理，常见的策略有重试（等一会儿再试，尝试其他分组服务，尝试其他机房服务，重试算法可考虑使用如指数退避算法），摘掉不存活节点（负载均衡/分布式缓存场景下）， 托底（返回历史数据/静态数据/缓存数据），等待页或者错误页。

对于非幂等写服务应避免重试，或者可以考虑提前生成唯一流水号来保证写服务操作，通过判断流水号来实现幂等操作。

对于有负载均衡的中间件，要考虑配置心跳/存活检查，而不是惰性检查。

超时重试必然导致请求响应时间增加，最坏情况下的响应时间=重试次数\*单次超时时间，这很可能严重影响用户体验，导致用户不断刷新页面来重复请求，最后导致服务接收的请求太多而挂掉，因此除了控制单次超时时间，也要控制好用户能忍受的最长超时时间。

超时时间太短会导致服务调用成功率降低，超时时间太长又会导致本应成功的调用却失败了，这也要根据实际场景来选择最适合当前业务的超时时间，甚至是程序动态自动计算超时时间。根据经验值，超时时间要设置的稍微大于tp99的响应时间。

客户端和服务端都应该设置超时时间，而且客户端根据场景可以设置比服务端更长的超时时间。如果存在多级依赖关系，如A调用B，B调用C，则超时设置应该是A>B>C,否则可能会一直重试，引起DDos攻击效果。不过最终如何选择还是要看场景，有时候客户端设置的超时时间就是要比服务端的短，可以通过在服务端实施限流/降级等手段防止DDos攻击。

在实际开发中，要重视超时时间，很多重大事故都是因为超时时间不合理导致的，设置超时时间只有好处没有坏处的。

超时和重试模式

超时（Timeout）可以避免线程的阻塞，从而可以规避级联的失效问题。

慢响应有很多种可能，有时候并不需要知道原因。这时只需要放弃，然后继续往下执行。客户端的超时因为远端服务端出现的问题，所以不能立即解决。再重试（Retry）还会是超时。所以，很多时候，缓存操作，事后再试是一个好主意。

### 5.7 服务容错模式

当业务不断扩张，集群的规模上千之后，每天发生服务调用的失败就成了大概率的事件。需要总结容错模式，以此来应对错误。

如图5.4所示，超时与重试模式，限流模式和舱壁隔离模式可以作为独立的模式。电路熔断器模式可以解决超时，限流和隔离的问题，保护服务自身和服务的下游，熔断之后可以使用回退模式做兜底的工作。

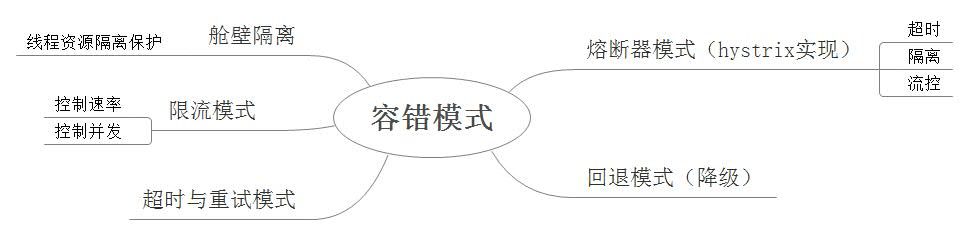


图5.4 容错模式

 电路熔断器模式

保险丝会在电流异常升高到一定的高度和热度的时候，自身熔断切断电流，从而起到保护电路安全运行的作用。同样，在服务集群中，如果调用的远程服务或者资源由于某种原因无法使用时，如果没有这种过载保护，就会导致请求的资源阻塞在服务器上等待从而耗尽系统或者服务器资源。很多时候刚开始可能只是系统出现了局部的、小规模的故障，然而由于种种原因，故障影响的范围越来越大，最终导致了全局性的后果。系统中的这种过载保护就是将要谈到的熔断器模式(Circuit Breaker)。

1.熔断实现机制

熔断器(Circuit Breaker)模式中有三种可相互迁移的状态，如图5.5所示。

闭合状态: 如果请求失败率没有超过阀值，或者关闭了熔断开关，则熔断开关处于closed状态，不进行降级处理。

打开状态：如果失败率超过阀值，或者打开了熔断开关，则启动熔断机制，调用降级处理方法进行降级处理。有异常后不会立即熔断，等失败率到了一定程度后，闭合状态才会迁移到开启的状态。

半打开状态：熔断开关打开后，不会一直熔断下去，会在一个时间窗口后进行重试，这就是半打开状态。

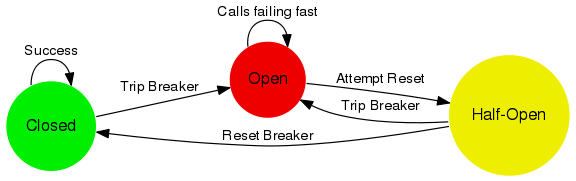


图5.5 熔断器状态迁移图

**2.**工程实现

Hystrix实现了快速失败和快速恢复，它会在一段时间内进行一次调用重试，重试成功就关闭熔断开关，否则还会处于打开状态。

以下几种错误会进入熔断状态：即信号量拒绝，线程池拒绝，失败（如异常），超时。

回退模式

在发生限流，熔断，超时重试失败的时候，为了马上恢复服务需要提供回退(Fallback)的机制，fail-silent,fail-fast,自定义方式是常见的回退策略.

失败沉默（fail-silent）一般用在可降级功能的场景下，数据为空也不会影响用户体验，直接返回缺省值或空值。

快速失败一般用在非强依赖的场景，如非核心服务超时的处理可以直接抛出异常。

自定义方式是如果业务是关键流程，并且严重影响用户体验的场景，可以使用备用服务获取数据。也可以将请求放入队列，待后续处理。可以使用默认数据，缓存来临时支撑。

1. 工程实现

Dubbo降级中的失败沉默，一般用于临时屏蔽出错的非关键服务。

比如：override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null

### 5.8 本章小节

本章首先讲了分布式服务框架非功能特性中容错实现的原理。

# 6 Hedwig服务框架设计和实现

### 6.1 Hedwig的模块和时序图

如图6.1,6.2所示为hedwig0.1,0.2版本的整体模块图。Hedwig-zk是zookeeper客户端，存储服务元数据到zookeeper中。Hewig-common是一些通用模块，包含一些框架底层抽象的接口和功能。Hewig-client模块是服务消费者，Hedwig-provider模块是服务提供者。Photon是0.2版本新增的一个模块，包括路由，负载均衡，连接池管理等功能。

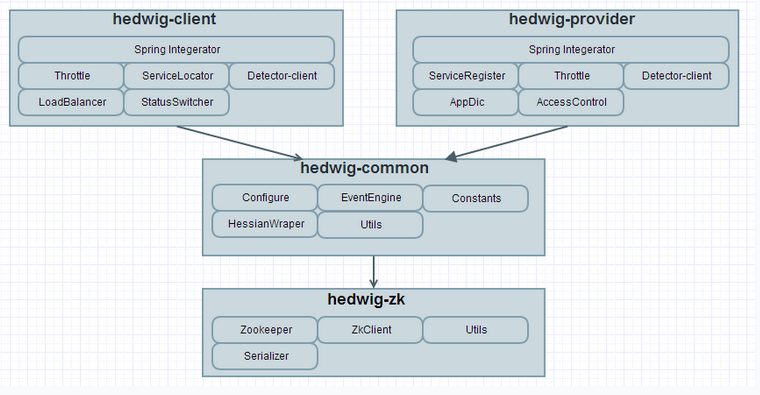
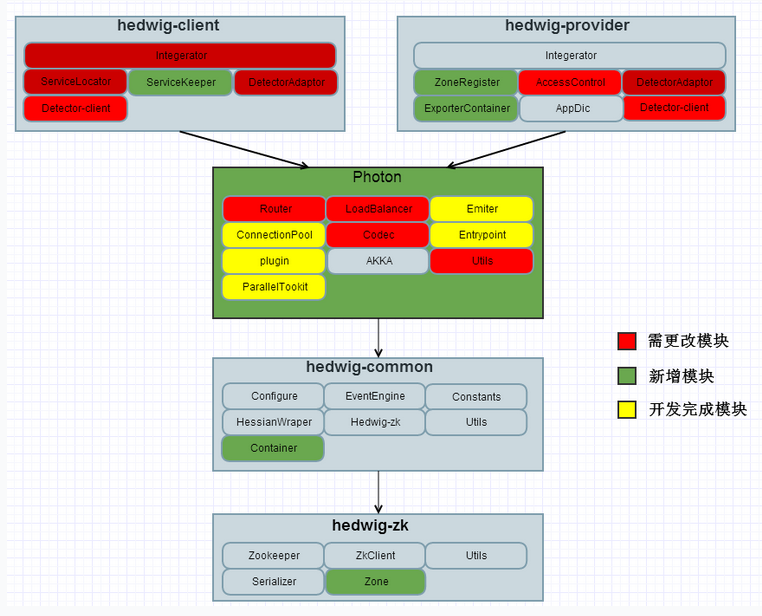


图6.1 0.1.x版本整体模块

图6.2 0.2.x版本整体模块

如图6.3,6.4所示为hedwig0.2版本的客户端和服务端各自的请求时序图。通过这些类和方法完成请求\响应过程。

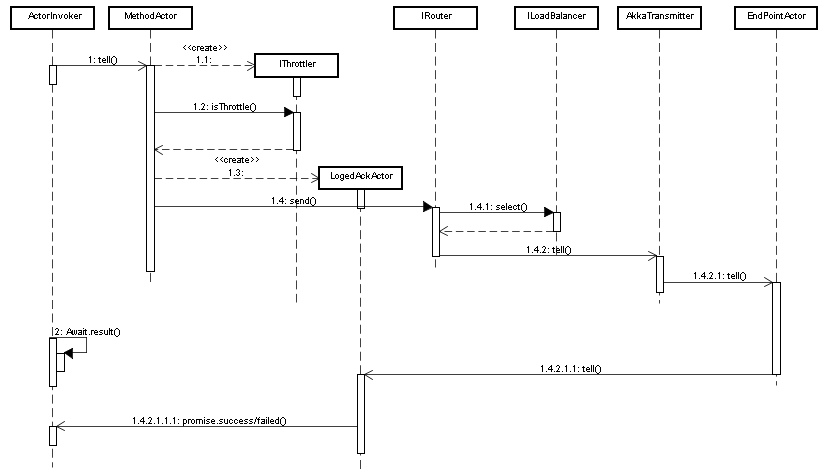


图6.3 0.2.x客户端请求时序图

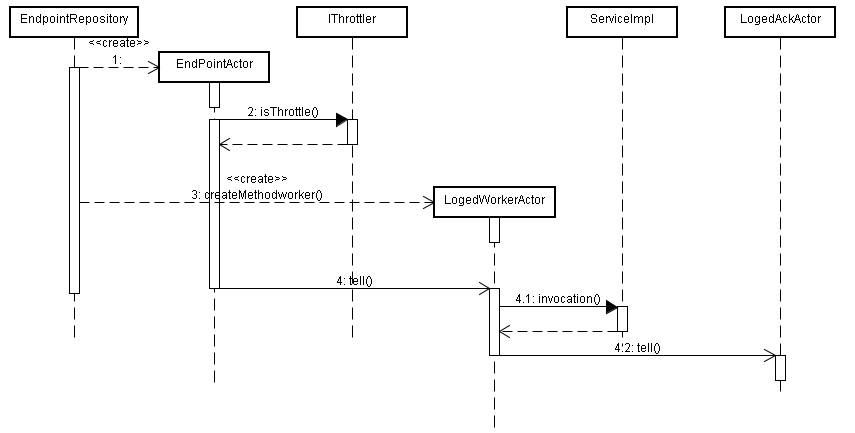


图6.4 0.2.x服务端时序图

### 6.2 Hedwig的功能模块设计和实现

1. Hedwig服务注册和发现功能设计

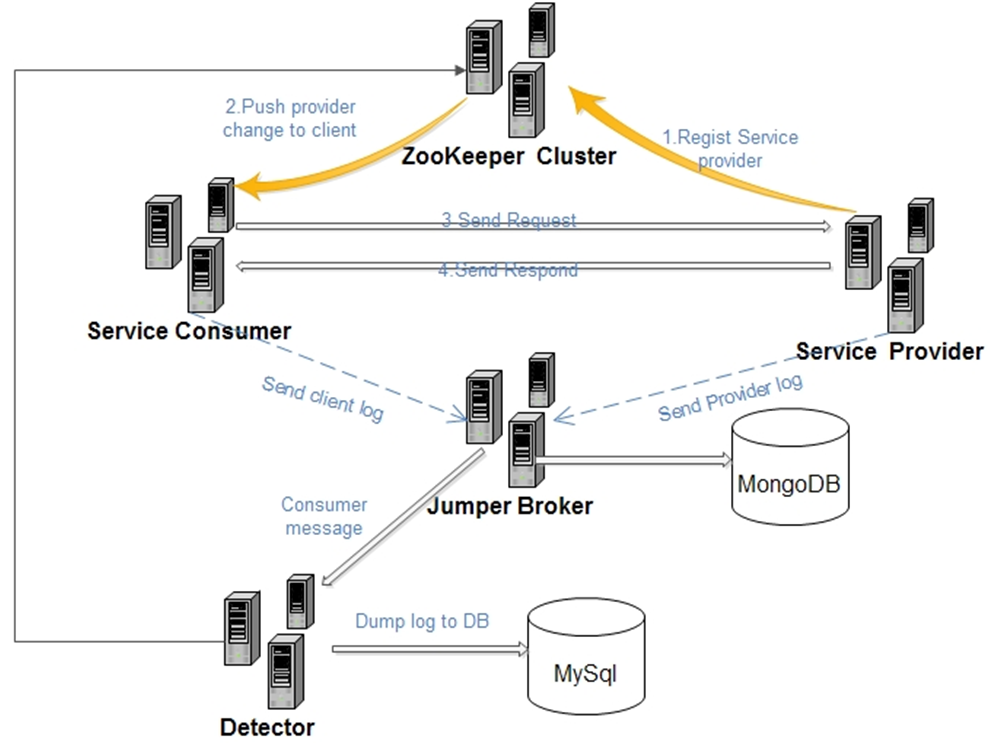


图6.5 Hedwig 服务注册和发现

Hedwig的客户端和服务端有服务自动发现机制, 如图6.5所示。

服务提供者在zookeeper中注册服务，往zookeeper的目录里写服务的元数据，使用的是zookeeper的临时节点。每个服务提供者正常启动之后会到Zookeeper集群进行注册，所谓注册就是在Zookeeper的树形结构中生成一个znode来标明服务可用。服务下线，链接断开，临时节点会自动消失。

服务消费者在zookeeper中查询到服务的元数据，然后调用服务,并且同时watch住了zookeeper的临时节点，订阅了这个服务。在Consumer端添加watcher监听的子节点变化，任何服务提供者节点的增减都会触发Locator模块重新获取可用服务器列表。并且将新列表传递给Load balance模块，为下次发送请求做准备。这样就可以感知到服务的下线和上线，做到了服务上线注册和下线去注册的功能。

通过zookeeper的协调，减少了客户端的配置，可以按需要添加和减少服务提供者的数量，极大的增加整体架构的弹性。

2. Hedwig动态负载均衡算法设计

switch (type) {

case ROUNDROBIN:

balancer = new RRBalancer(routees);

break;

case CONSISTENT\_HASH:

balancer = new CHBalancer(routees);

break;

case BROADCAST:

balancer = new BCBalancer(routees);

break;

case LEAST\_USAGE:

balancer = new LUBalancer(routees);

break;

case WEIGHT\_ROUNDROBIN:

default:

balancer = new WRRBalancer(routees);

break;

}

图6.6 Hedwig 负载均衡算法

如图6.6所示, Hedwig实现了Round Robin，基于权重的Round Robin，一致性hash，广播算法。默认使用基于权重的轮询算法.

所有路由算法都以可用节点生成的环为基础，每次通过适当的方式从环中取一个或多个节点来发送请求。RoundRobin算法中，每次都取环中的下一个节点.WeightRoundRobin算法中，在构造环时根据节点的权重来决定环中节点的冗余个数。广播算法中，一次给环中的每个节点都发送请求.

如图6.7所示，能看到3个服务节点的调用量比较小，说明使用的负载均衡算法使得调用量不是很均匀,所以可以重试其他的负载均衡算法。



图6.7 负载均衡监控

3. Hedwig服务路由设计

 基于IP路由

Hedwig客户端里可以配置target属性，表示指定访问机器IP，这个功能主要是给用户在本地开发测试时用的。

基于分组路由

如图6.8所示, 每个Pool的分组方式由服务提供方在发布客户端时定义，可以根据调用方需要的功能或重要级别加以区分。服务发布后把服务端节点分入预先定义的分组中，同一个节点可以属于多个分组。请求只会路由到指定分组中的节点。

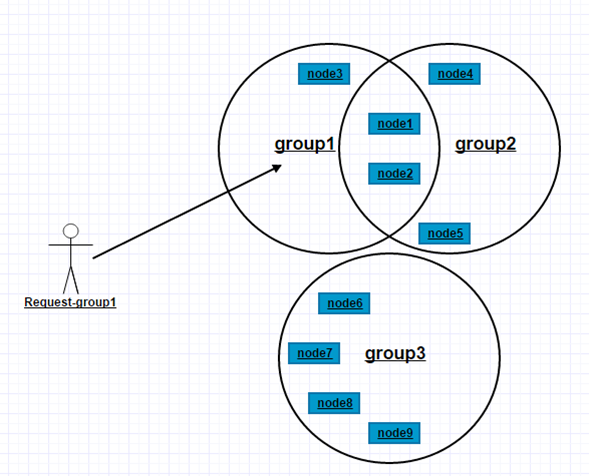


图6.8 客户端和分组的关系

Hedwig的分组可以在Detector里管理，并且可以在分组里添加机器。如图6.9,6.10所示。

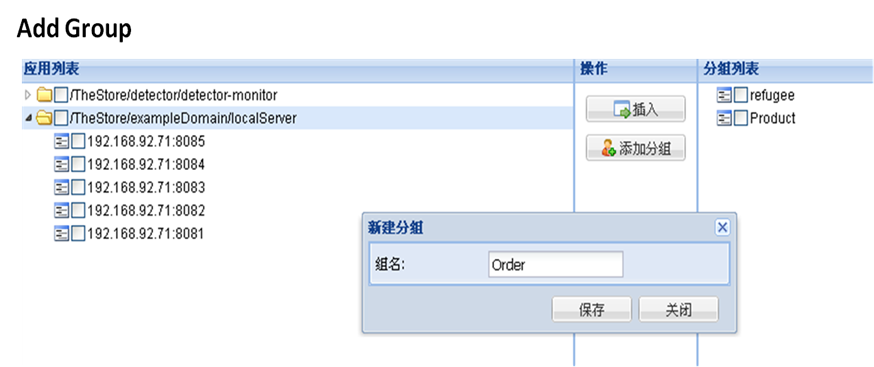


图6.9 在Detector中添加分组

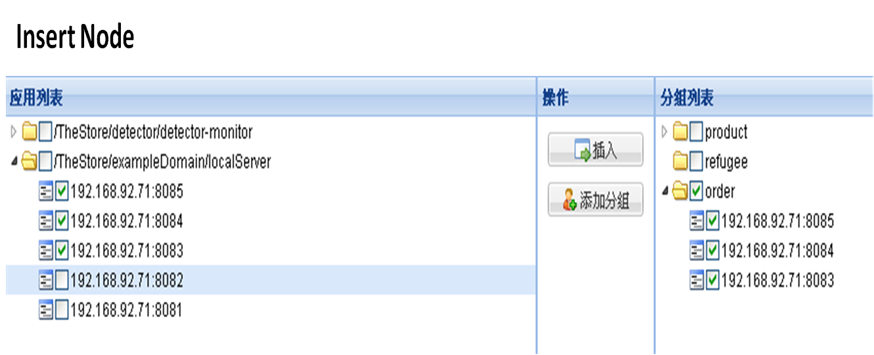


图6.10 在分组中添加机器

4. Hedwig序列化和反序列化实现

Hedwig分自身的序列化和可扩展的序列化，自身的序列化也是默认的序列方式，用户也可以继承HedwigCompatibleCodecProcessor自定义序列方式。如图6.11,6.12所示。

public class HessianCodecProcessor implements ICodecProcessor<RemoteRequest<InvocationContext>, RemoteResponse>{

private static Logger logger = LoggerFactory.getLogger(HessianCodecProcessor.class);

public HessianCodecProcessor() {

byte[] bytes = this.toBinary("init");

Object s = this.fromBinary(bytes);

}

@Override

public byte[] reqeust2binary(RemoteRequest<InvocationContext> request) {

return toBinary(request);

}

@Override

public RemoteResponse binary2respones(byte[] array) throws Throwable {

Object o = fromBinary(array);

return o == null ? null : (RemoteResponse) o;

}

@Override

public byte[] respones2binary(RemoteResponse response) {

return toBinary(response);

}

@Override

public byte[] error2binary(String code, String msg, Object o) {

return null;

}

@Override

public RemoteRequest<InvocationContext> binary2request(byte[] array) {

Object o = fromBinary(array);

return o == null ? null : (RemoteRequest<InvocationContext>) o;

}

}

图6.11 hessian序列化

public class HedwigCompatibleCodecProcessor implements ICodecProcessor<RemoteRequest<InvocationContext>, RemoteResponse> {

@Override

public byte[] reqeust2binary(RemoteRequest<InvocationContext> request) {

...

out.call(request.getMangleName(), request.getParameters(), request.getContext());

return array;

}

@Override

public RemoteRequest<InvocationContext> binary2request(byte[] array) {

...

HedwigHessianInput in = new HedwigHessianInput(bis);

o = in.readObject();

return (RemoteRequest<InvocationContext>) o;

}

@Override

public byte[] respones2binary(RemoteResponse obj) {

ByteArrayOutputStream baos = new ByteArrayOutputStream();

HedwigHessianOutput out = new HedwigHessianOutput(baos);

array = baos.toByteArray();

return array;

}

@Override

public RemoteResponse binary2respones(byte[] array) throws Throwable {

...

resp = o != null ? ResponseFactory.createSuccessResponse(o, 0) : null;

...

return resp;

}

@Override

public byte[] error2binary(String code, String msg, Object exception) {

ByteArrayOutputStream baos = new ByteArrayOutputStream();

...

array = baos.toByteArray();

return array;

}

}

图6.12 可扩展的序列化

5. Hedwig服务调用设计(0.2 Akka版本)

public static final String CALL\_SYNC = "syncPool";

public static final String CALL\_ONEWAY = "oneway";

public static final String CALL\_FUTURE = "future";

图6.13 akka的三种调用方式

Hedwig Akka有三种调用方式，如图6.13所示。

CALL\_SYNC是同步的方式，使用Await.result异步转同步。CALL\_ONEWAY是异步单向的调用方式。CALL\_FUTURE是异步future的调用方式。

6.Hedwig并行服务调用实现

并行调用实现如图6.14所示：批量服务调用加上Future机制。并行调用比串行调用效率高。实现中没有产生新的线程，使用者不用担心依赖线程上下文的功能会抛出异常。

final IQueryService queryService = (IQueryService) context.getBean("queryService2");  
Map<String, List<Future<RemoteResponse>>> fMap = ParallelToolkit.*createScalaFuture*(new MultiCall<RemoteResponse>() {  
 @Override  
 public void call() throws Throwable {  
 for (int i = 0; i < 10; i++) {  
 queryService.queryStrings(Long.*valueOf*(i), *s*);  
 }  
 }  
});

//do other;  
for (int i = 0; i < 10; i++) {  
 Result r = (Result) ParallelToolkit.*getResult*(fMap.get("queryService2.queryStrings\_long\_string").get(i), 1000);  
 System.*out*.println("====================Result:" + r.getValue());  
}

图6.14 Hedwig并行接口

并行调用比串行调用减少了更多的响应时间。如图6.15所示，如果Task1耗时5ms，Task2耗时7ms，Task3耗时8ms，串行调用需耗时5+7+8=20ms，并行调用只需要消耗Max（5,7,8）=8ms。

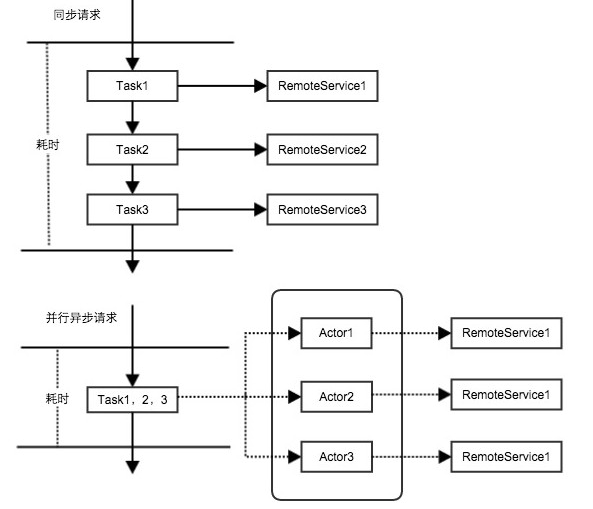


图6.15 服务串行和并行调用比较

7. Hedwig可靠性设计

用netty做底层rpc通信的服务框架的心跳机制靠netty的空闲链路检测机制实现，链路关闭后会触发事件，重新发起连接操作。整个可靠性是依赖netty实现的。而Hedwig0.2.x是靠akka框架本身实现的心跳检测/重连机制来保证链路的可靠性。

8. Hedwig服务框架部署方式设计

Hedwig 0.1.x Hessian版本中, 分布式服务框架与应用一起打包,它作为应用的一个依赖包，web容器是tomcat，详见图3.6。

Hedwig 0.2.x Akka版本把服务框架自身需要变为一个容器来承载远程服务,akka底层使用netty作为通信框架，详见图3.8。

9. Hedwig协议栈设计

public enum ProtocolType {

NONE("NONE"),

HTTP(SupportedProtocol.HTTP),

HTTPS(SupportedProtocol.HTTPS),

TCP(SupportedProtocol.TCP),

UDP(SupportedProtocol.UDP),

AKKAtcp(SupportedProtocol.AKKAtcp),

AKKAudp(SupportedProtocol.AKKAudp);

}

图6.16 Hedwig支持的协议

Hedwig本身支持基于Hessian的HTTP共有协议和基于Akka的TCP私有协议，如图6.16所示。HTTP共有协议是Hedwig0.1版本使用的，TCP私有协议是Hedwig0.2版本使用的。

### 6.3 Hedwig功能组件设计

hedwig和其他主流服务框架的功能点和模块有共通点，抽象出设计模式，按模块的职责分，可以分成以下几个组件，如图6.17所示。

1. Client Proxy由HedwigEventInterceptor实现，Requestor称为Engine层，BaseHandler称为handler层。

2. HedwigEventInterceptor调用Engine层，Engine层调用handler层。

3. Invoker由HedwigHessian2SkeletonInvoker实现，Server Request Handler由HedwigHessianSkeleton实现。

4. Remote Error分别由EngineException和HandlerException实现。

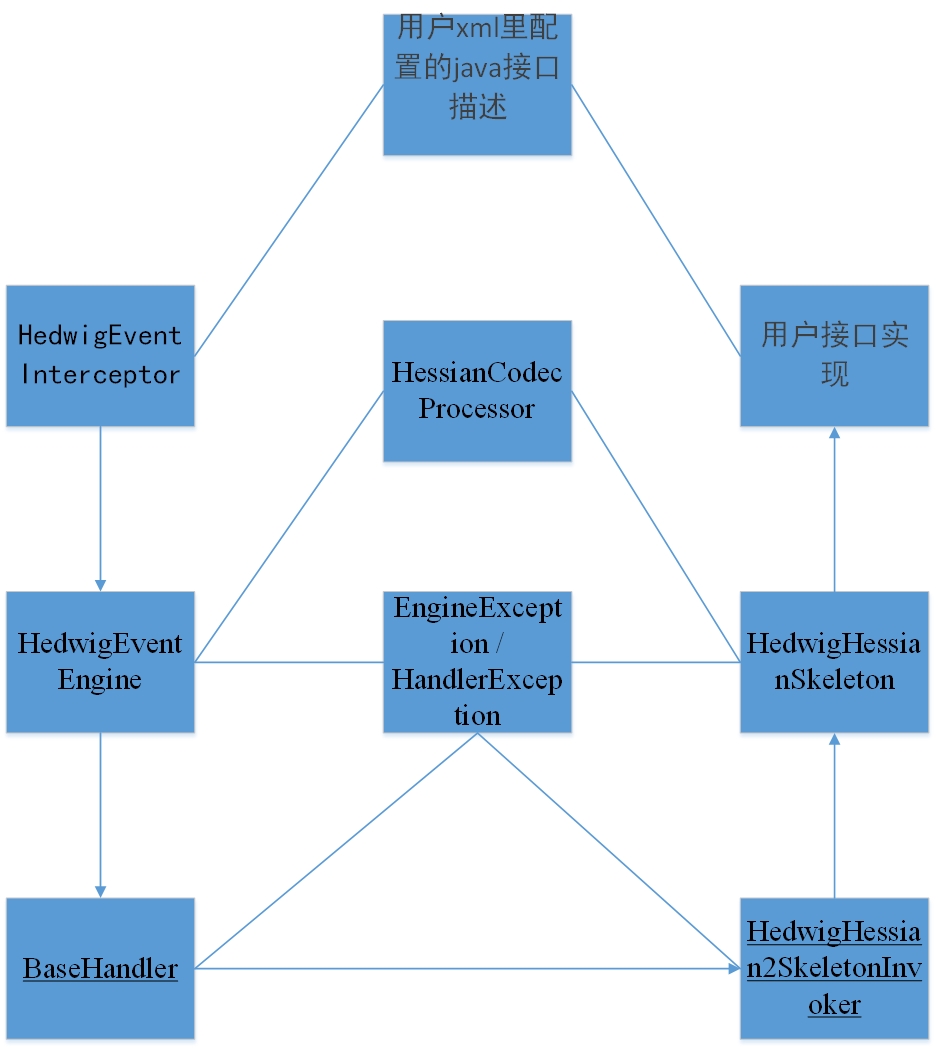


图6.17 Hedwig组件关系

### 6.4 hedwig容错设计和实现

 Hedwig Failover实现

当一个业务请求碰到服务一个故障节点时候报错，我们把故障节点剔除，并直接发返回异常给上层业务方。这种场景，为了减少业务系统报错，增强系统可用性，需要做一种优化，需在上层业务所能接受的时间内，进行重试另外一个节点。并将正确结果返回上层业务系统。

需要设置重试开关redoAble，可重试服务必须为幂等服务，触发条件为非业务异常。在业务方配置的超时时间内进行重试，最多重试3次，重试时选择非当前出错节点。如图6.18为failover流程。

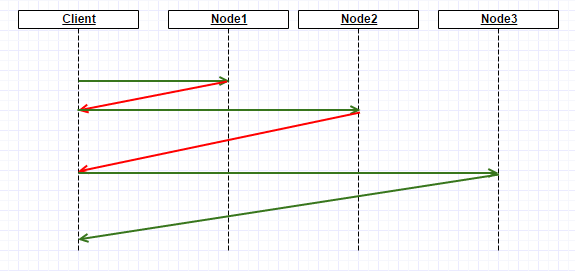


图6.18 Hedwig failover流程

如图6.19所示， 如果客户端中收到的响应是ReceiveTimeout的时候会调用retryMessage方法重试其他可以的节点。服务端慢了超时了，会采用failover机制。

} else if (message.equals(ReceiveTimeout.getInstance())) {

Channel channel = \_msg.getSenderChannel();

if (channel != null) {

ClientNettySystem.getInstance().getSyncConnPool().destory(channel);

}

RouteeRef routeeRef = \_request.getRoutee(\_seq);

RouteeWraper routee = routeeRef != null ? routeeRef.getRoutee() : null;

if (routee != null) {

PhotonStatus status = routeeRef.getStatus();

if (PhotonStatus.ENABLE.equals(status)) {

routee.setTmpDisable();

logger.error("Request read timeout on:" + routee.getHostUrl() + ",kick out node and msg going to be retry");

}

} else {

logger.error("Request don't assigned a remote host yet,sequence:" + \_seq);

}

HedwigMonitorClientUtil.setException(cbLog, RemoteExceptionFactory.createRequestTimeoutException(\_request));

cbLog.setInParamObjects(\_request.getParameters());

retryMessage(cbLog);

图6.19 Hedwig failover实现

 Hedwig Failcache实现

如图6.20所示，当服务端有网络异常并且客户端有限流的时候，客户端会设置服务为TEMPORARY\_DISENABLE状态，表示暂时不可用.

if (HandlerUtil.*isNetworkException*(e) && context.getClientProfile().isClientThrottle()

&& !sp.getCurStatus().equals(ServiceStatus.*TEMPORARY\_DISENABLE*)) {

sp.setCurStatus(ServiceStatus.*TEMPORARY\_DISENABLE*);

logger.warn(InternalConstants.*LOG\_PROFIX* + sp.getHostString() + " has kickout of candidate!!!");

}

图6.20 服务状态迁移

如图6.21所示，服务在正常运行时，服务会在Enable状态。如果这个时候有网络的异常，客户端设置服务进入TEMPORARY\_DISENABLE（临时不可用的状态）。 经过relive的策略后， 服务可以从TEMPORARY\_DISENABLE状态迁移到Enable状态。

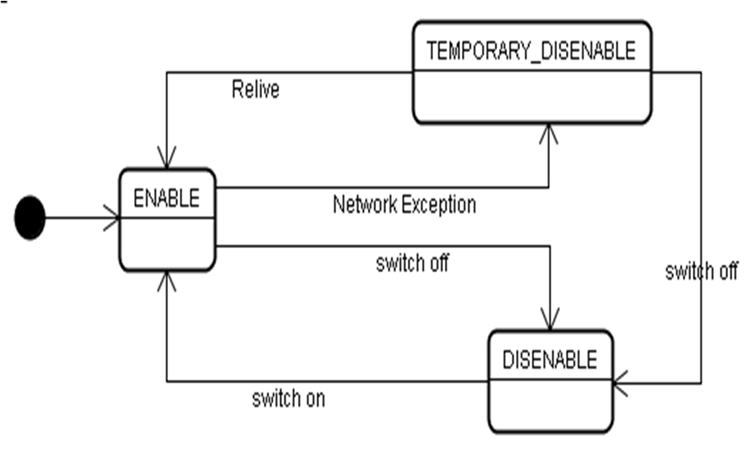


图6.21 Hedwig服务状态迁移图

如图6.22所示，触发临时不可用状态的网络异常包括链接的Connect异常,Socket异常,IO异常,Socket超时异常。

static{

*NETWORK\_EXCEPTIONS* = new HashSet<String>();

*NETWORK\_EXCEPTIONS*.add(ConnectException.class.getName());

*NETWORK\_EXCEPTIONS*.add(SocketException.class.getName());

*NETWORK\_EXCEPTIONS*.add(IOException.class.getName());

*NETWORK\_EXCEPTIONS*.add(java.net.SocketTimeoutException.class.getName());

}

图6.22 4种网络异常

自动复活(relive)的策略有两种。时间策略是第一次被踢出会在500ms后恢复，如果复活后的请求仍然失败，下次复活时间x2,时间上限为60s。检索次数策略是被踢出节点被检索次数达到50次后恢复，如果复活后的请求仍人失败，下次复活需要的检索次数x2，次数上限60000。混合策略及tps策略是当tps小于100时会触发时间测率，大于100会触发检索次数策略。

复活时间\*2和检索次数\*2来保证服务端有一段时间来恢复到正常的状态，如果大量客户端一直重试会对服务端有很大的压力，反而对恢复流程不利。

在自动复活的时候，实际这是一个failcache的容错策略，会重试有问题的节点。

 Hedwig人工降级设计

如图6.23所示，在detector里实现了对Hedwig服务的人工降级开关。

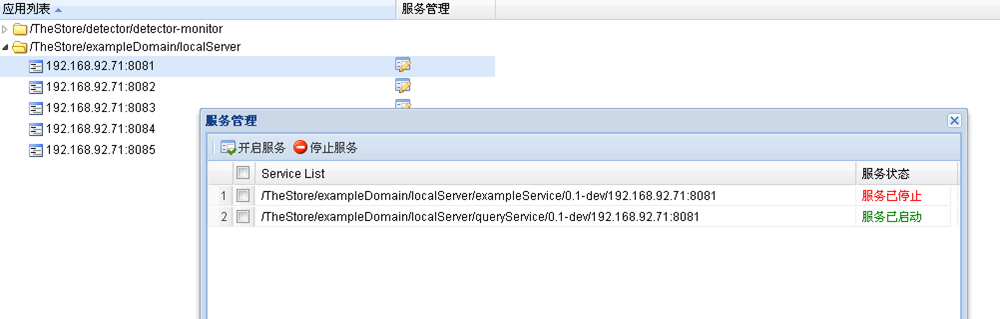


图6.23 在detector里控制服务的启停

如果服务已经开启,可以在detector里关闭服务,客户端会自动感知到服务状态。

如果服务已经关闭,可以在detector里开启服务,客户端会自动感知到服务状态。

 Hedwig集群隔离设计

当一个服务集群提供众多pool使用时，服务资源是有限的，生产环境下，由于业务特点，不同pool对服务集群资源消耗是不一样的，有的非常均衡，有的非常突变。有时一个使用方由于bug或其它什么原因，突然大量占用服务资源，在整个服务集群造成影响，我们需要将这种影响做隔离，或者说将影响缩小在局部范围，不要扩散到整个集群。需要根据不同调用方业务特点，对服务进行分组。

如图6.24所示，实现了服务集群的分组，客户端可以调用分组内的机器。如果一个分组内机器不稳定，不会影响到别的分组。

<bean id="baseHessianService"

class="com.yihaodian.architecture.hedwig.client.HedwigClientFactoryBean">

...

<!--当有分组需求时配置-->

<property name="groupName" value="testGroup2"></property>

...

</bean>

图6.24 服务分组配置

 Hedwig线程隔离设计

如图6.25所示，Hedwig akka是一种细粒度的隔离, 服务分组中每个服务有一个独立的线程池。methodWorkerCount表示线程池中工作线程的数量。 用户可以自己设置，默认值是16。

一个actor对应一个mailbox。worker线程池管理多个actor的mailbox, 拿一些消息之后, 会处理这个消息。如果消息有积压，可以伸缩这个线程池，产生或释放更多线程。

<bean name="readService" class="com.yhd.arch.laserbeak.provider.SpringServiceExporter">

<property name="appMeta" ref="appMeta"></property>

<property name="service" ref="rs" />

<property name="serviceInterface" value="com.yhd.arch.spi.IReadService" />

<property name="serviceVersion" value="0.1-jl"></property>

<property name="methodWorkerCount" value="50"></property>

<property name="initStart" value="true"></property>

</bean>

图6.25 服务工作线程池的配置

 Hedwig限流接口的总请求数设计

Hedwig akka客户端流控:限制某个方法的总请求数，如图6.26所示。

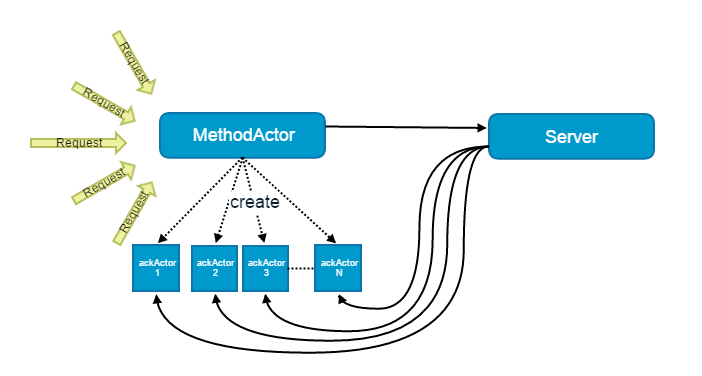


图6.26 Hedwig methodActor和AckActor的关系（客户端流控）

一个服务由多个方法方法构成，一个方法会对应一个methodActor，发送一个请求后，methodActor会对应产生一个AckActor，AckActor表示对请求的发送/接收处理。一个接口发送了多少消息等待处理，就会有多少个AckActor。

所以，hedwig客户端根据methodActor下AckActor的个数也就是总请求数做流控，如果请求量大于阀值，就拒绝后续请求，配置如表6.1所示。

表6.1 Hedwig akka客户端配置

|  |  |
| --- | --- |
| **Option and Default Value** | **description** |
| photon.emmitor.channel.count | core size of connection pool for upward compatible |
| photon.emmitor.channel.max | max size of connection pool for upward compatible |
| photon.emitter.throttler=  com.yhd.arch.photon.core.MethodActorThrottler | method throttler class name |
| photon.emmitor.method.limit=800 | pending request count in client |
| photon.emmitor.relive.interval=100 | first relive interval, interval\*2 next fail |
| photon.emmitor.relive.interval.max=60 | max relive interval |
| photon.emmitor.relive.count=10 | first relive count,count\*2 next fail |
| photon.emmitor.relive.count.max=60000 | max relive count |

 Hedwig限流接口的时间窗请求数设计

Hedwig akka服务端流控,如表6.2所示:根据某个时间窗口中请求数tps的阀值做流控。

表6.2 Hedwig akka 服务端配置

|  |  |
| --- | --- |
| **Option and Default Value** | **Description** |
| photon.remote.endpoint.receiver.count=100 | server entry actor count |
| photon.endpoint.throttler=  com.yhd.arch.photon.plugin.EndpointThrottler | throttler class name |
| photon.endpoint.throttle.enable | switcher of server throttle, value is disable/enable |
| photon.endpoint.throttle.tps=3000 | server throttle tps |
| photon.endpoint.throttle.interval=3000 | throttle calcuate interval |

 Hedwig超时与重试机制设计

如图6.27所示，Hedwig客户端可以设置两个超时相关的参数(timeout和readTimeout)

1.readTimeout的值为服务端处理请求的时间+网络传输时间；

2.timeout的值根据准许的重试次数可以设置为readtiemout的3-5倍.

3.Timeout = 重试次数 \* readTimeout.

<bean id="queryService\_AKKA\_2" parent="laserBeakParent\_2">

...

<!-- 创建并维持网络连接所允许的最大等待时间, 单位为毫秒. readTimeout=服务端处理请求的时间+网络传输时间 -->

<property name="timeout" value="50000"></property>

<!-- 从发起请求到收到响应所允许的最大等待时间,单位为毫秒.timeout的值根据准许的重试次数可以设置为readtiemout的3-5倍 -->

<property name="readTimeout" value="200000"></property>

<!--当服务调用出错时, 是否允许重试.-->

<property name="autoRedo" value="true"></property>

...

</bean>

图6.27 Hedwig 超时时间

### 6.5 Hedwig测试与性能

Hedwig实现的版本分为两个0.1. x和0.2.x， 0.1.x基于hessian协议， 0.2.x基于akka框架。Hedwig0.2.x选用了Akka框架提供的actor模型，它是普遍的并发模型之一，它提供了全异步化的功能，有更高的吞吐，更容易的编程模型。Hedwig整体性能见图6.28。

在akka的类库上做了一个photon项目， 其中序列化是并行的， 相对于hessian有更高的性能。

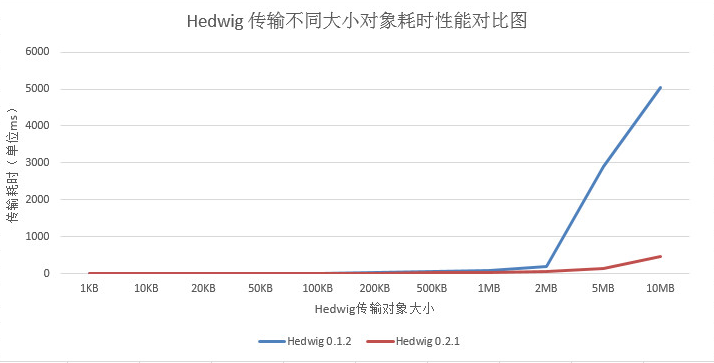


图6.28 Hedwig整体性能

1. 当传输对象大小100KB以内，0.2版本和0.1版本性能基本相当。

2. 当传输对象大小100KB以上时，0.2版本比0.1版本性能要优越。

3. 当传输对象大小2MB左右，0.2版本性能优势十分明显。

结论: 在相同大小对象传输上，0.2版本性能要优于0.1版.在吞吐持续变大的情况下，hedwig0.2版的响应时间没有变的太大，也就是说在单位时间内，hedwig0.2版能承受更大的TPS并发量。所以，hedwig0.2版为公司业务团队提供了一个高并发，吞吐量都很高的分布式服务框架。

### 6.6 本章小节

本章首先论述了Hedwig功能模块、功能组件、容错的详细设计和实现。最后论述了hedwig的测试和性能，验证了hedwig是一个高并发，高吞吐的分布式服务框架。

# 7 服务监控

### 7.1 监控的种类

及时发现问题是解决问题的前提。及时发现问题的手段是完善监控与报警，包括从硬件、网络，到应用业务的监控。服务监控从底层硬件到上层应用服务一般有以下的三个层次：

 基础设施监控

监控会探测硬件瓶颈相关的早期迹象并捕获硬件故障信号，在出现更大的问题之前对其进行应对。

基础设施监控的范围包括机器的健康状况、CPU使用、内存消耗以及网络带宽，基于这些监控信息，能够判断基础设施的当前状态，从而进行报警或者必要的升级扩展。

 应用级别监控

应用级别的监控涉及到监控各种服务器的状态，如应用服务器、数据库服务器等，要监控的参数则是与应用或工具相关的。要监控的指标包括JVM内存，JVM GC情况，线程数，磁盘IO，索引的读取/写入操作。

 服务监控

服务监控就是要监控每个服务的吞吐量和性能，进而确保在任何时间都能满足SLA的需求。

这种类型的监控一般都需要对应用进行Instrument操作，让Instrumentation是可配置的，通过收集器（collector）收集应用的状态，并阶段性地将这些状态发送到永久存储、分析器和预警系统中。

此类监控往往会产生大量的数据，因此有可能会影响到性能。

### 7.2 Hedwig监控

 Hedwig基础设施监控

Hedwig异常报警如图7.1所示,表示某个服务节点的出流量异常。

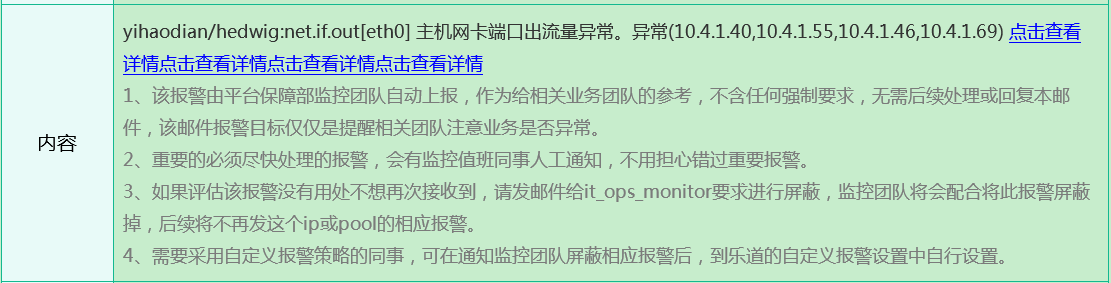
****

图7.1 Hedwig机器网络出口流量异常报警

Hedwig应用级别监控

一号店乐道平台和一号店zabbix监控系统的应用级别监控包括以下这些指标，如图7.2所示：JVM(gc,线程)，cpu，内存，网卡/磁盘读写，tomcat线程池，连接数等。

可用性监控主要观察系统是否存活，服务可用率是否低于阀值。如图7.3所示，Zookeeper节点，hedwig节点，detector节点宕机或者资源不足，会有报警邮件。

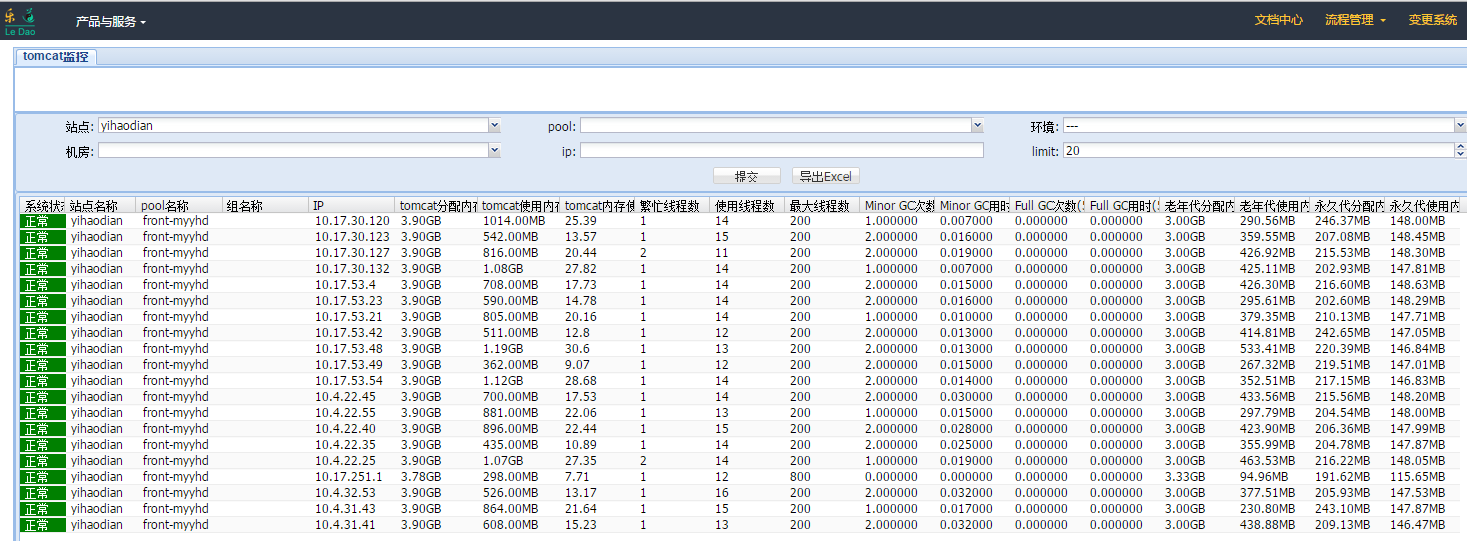
****

图7.2 乐道里Hedwig应用监控

cid:image001.png@01D2BDCB.4A4544B0图7.3 hedwig服务 zookeeper集群报警

 Hedwig服务监控

服务监控可以分为对所有服务调用的整体指标的监控和对单个服务\方法处理能力的监控。

1. 处理能力的监控

如图7.4,7.5,7.6,7.7所示，Detector对应用的性能和健康度监控, 监控的指标有服务调用的成功率/失败率/响应时间/调用量。问题排查时可以查异常调用的错误日志和明细日志，当大促错误量比较大的时候会做错误日志的抽样，保证日志量不会太多。服务调用链路图能看到整个链路的全貌和服务调用的情况，也能看到单个链路上细粒度(pool名/服务名/方法名/分组名)的调用情况。



图7.4 Detector服务治理

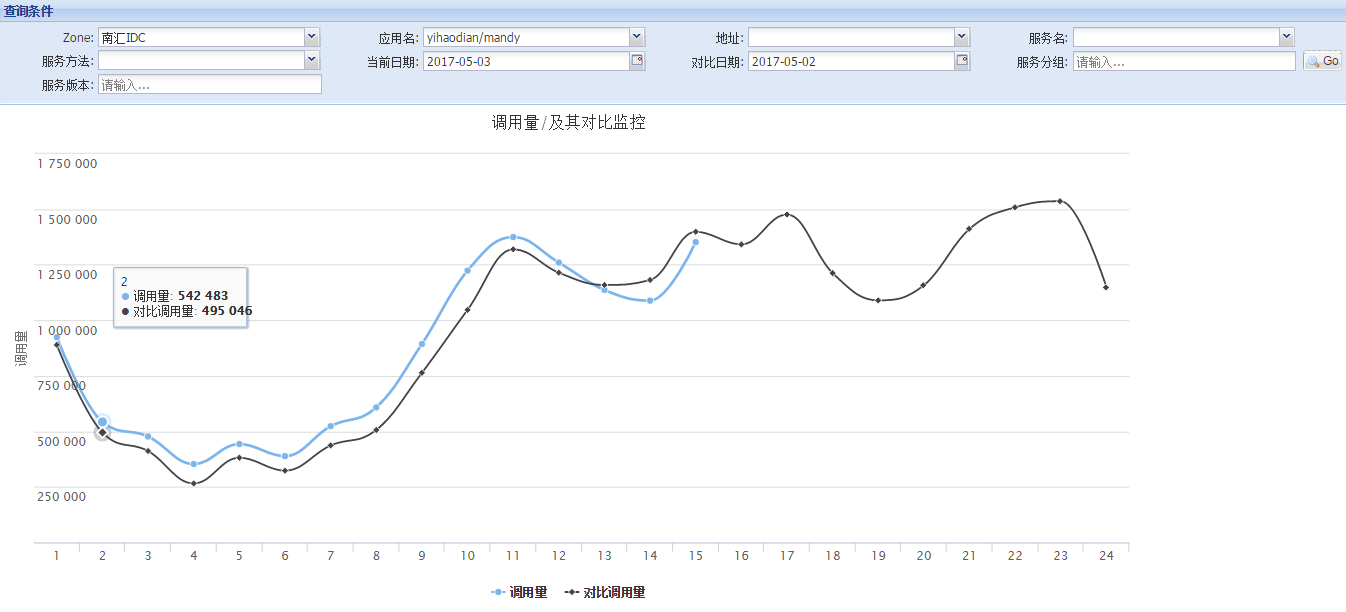


图7.5 服务每小时调用量

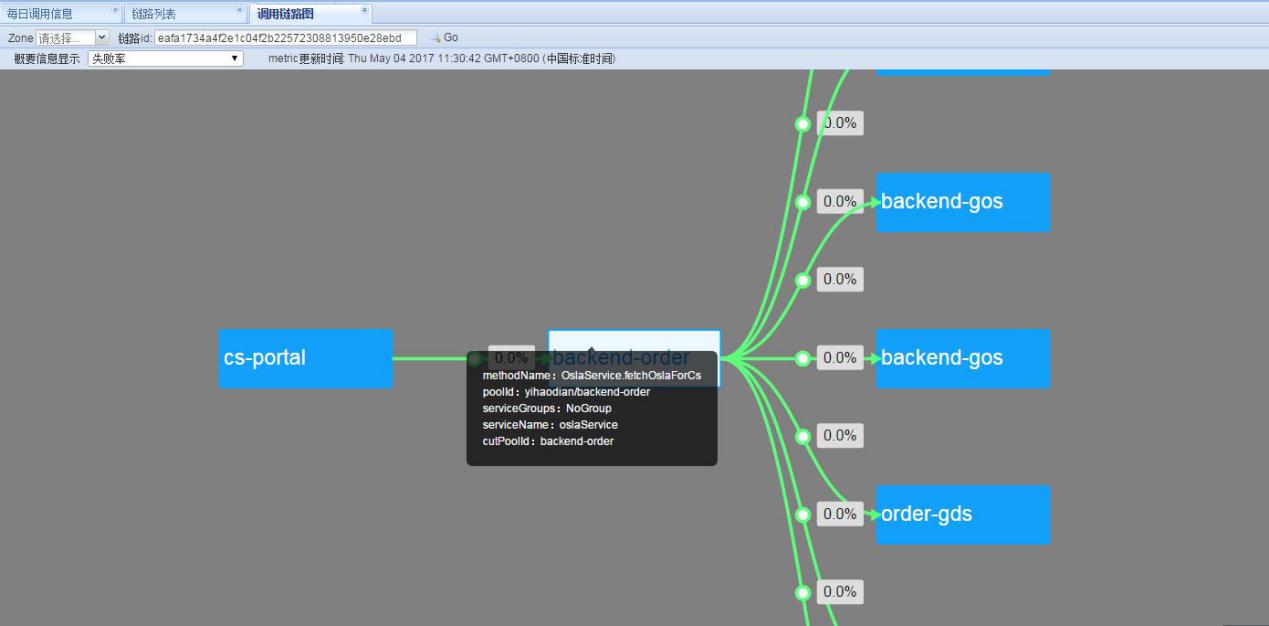


图7.6 detector链路调用关系

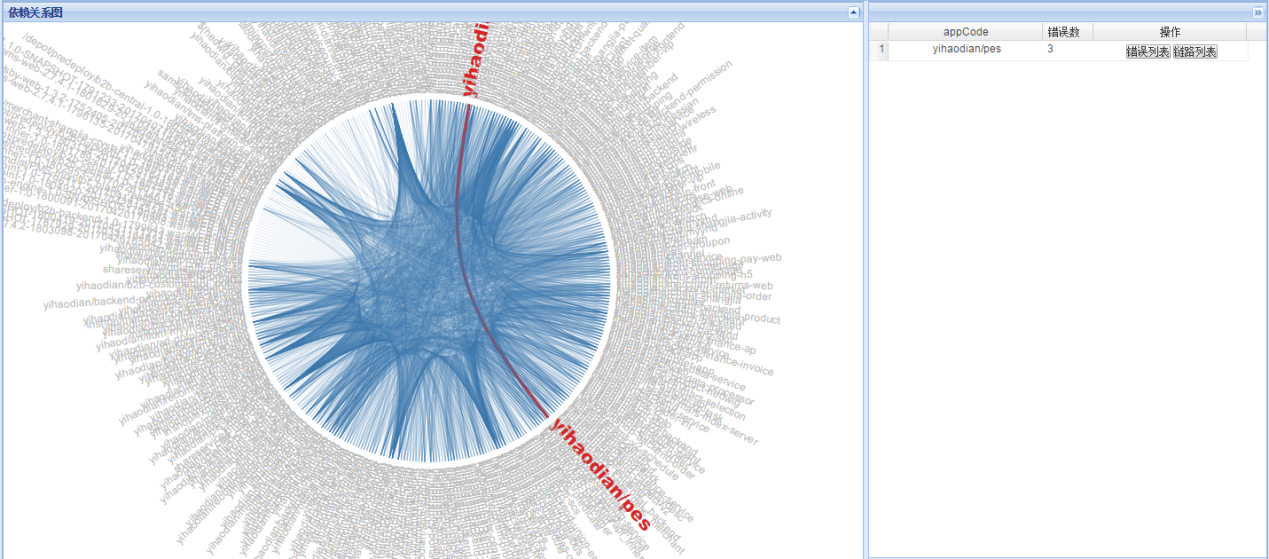


图7.7 服务pool之间的依赖关系图

1. 整体指标监控

如图7.8所示，现在,一号店的Hedwig南汇IDC每天有1200个服务的31亿多次的调用，成功率为99.99%以上，平均耗时在19ms左右，包括170个左右的pool。

Hedwig IDC中服务整体指标包括整体的调用量，成功率，平均耗时。

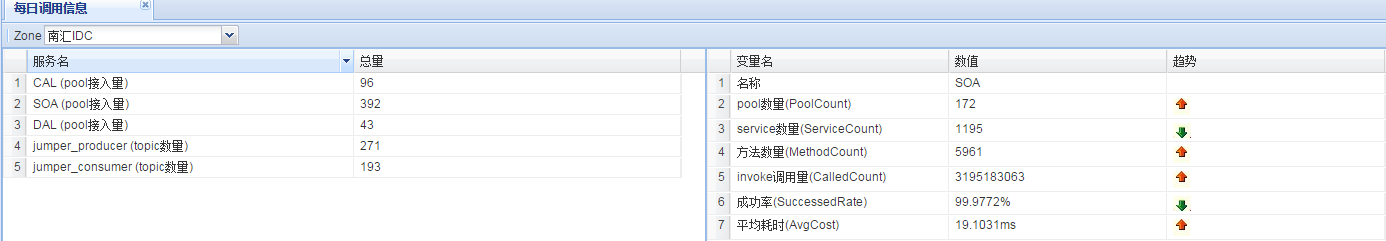


图7.8 一号店南汇IDC中服务整体的每天调用量

### 7.3 服务监控上的应用

业务往往是很复杂的，这样服务和服务之间相互依赖的关系也变的很复杂，服务链路监控中能看到服务之间的调用关系和服务的分层关系，这样就能看到业务全局的视图。

为了应对大促做系统的扩容，有些业务变更需要做系统的缩容，监控中能看到服务当前的负载和变化情况，并给予扩容或者缩容以指导。

 功能性依赖梳理

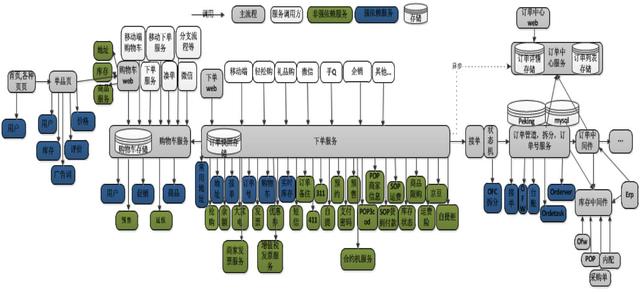


图7.9 商城交易平台的服务依赖图

有了监控中的链路调用关系，就可以看到整个公司业务的全貌。如图7.9所示，这是商城交易平台的服务依赖图，绿色部分是弱依赖的服务，蓝色部分是强依赖的部分,白颜色的是服务调用方，灰颜色的是主流程。

强依赖的一般是不能降级的核心服务。比如实时库存，购物车，商品服务，订单号服务等。弱依赖的一般是可以降级的非核心服务。比如优惠券，运费险，自提服务，订单备注等。

弱依赖的部分如果有问题，可以在detector里面做手工开关降级，或者是自动降级。

 容量规划

数据在线上主系统中存留的时间是主要考虑的因素。出现高峰时，允许积压数据量和积压时间,正常情况下可以慢慢处理.

在异常情况下，有个服务强依赖下游系统，并且下游系统被压垮之后，其容量发生了变化，如果原来容量是5，压垮之后容量会减小，可能变成了3。

在大促时必然会对系统造成很大的压力，当分配的量大于了单机的容量，如果系统没有做好保护，持续一段时间后，系统将不能很好的恢复到原来的状态，即使把分配进入的流量减少到系统容量以下也不能快速恢复。

容量评估时会评价一个水位，保证大于这个水位会有限流或降级来保护下游系统。

### 7.4 本章小节

本章首先论述了3个层次的监控，然后讲述了hedwig在这3个层次的监控上实现了哪些功能，最后论述了服务治理中的功能性依赖梳理和容量规划。

# 8 总结和展望

从12年开始一号店就开始了公司整体服务化的进程，其中最主要的产品之一是服务治理框架hedwig。hedwig几乎承载了公司商城网站和公司内部所有的业务。Hedwig在这几年的发展中也是在各种大促和大流量的情况下经受过大考，一致得到了公司的认可。

Hedwig的核心的功能和非功能特性已经实现了，但和阿里的dubbo，唯品会的OSP，点评的pegioin相比，还是有可以借鉴的地方，有些算法和细节还有精雕细琢的空间。

服务调用错误每天都在发生，错误模式是很好的一个总结，因此服务框架设计和实现需要参考这部分的内容。

hedwig监控方面的功能基本都已经实现了，但在APM相关的全链路服务监控还比较薄弱，这是以后要加强的一个方向，能够更快速响应错误的服务调用。

# 参 考 文 献

［1］Volter M. Remoting Patterns - Foundations of Enterprise, Internet and Realtime Distributed Object Middleware[J]. IEEE Computer Society, 2006, 8(6):60-68.

［2］Brewer E. CAP twelve years later: How the "rules" have changed[J]. Computer, 2012, 45(2):23-29.

［3］ Frank Buschmann ， Keclin Henney, Douglas C.Schmidt (作者) 肖鹏 陈立 (译者)．面向模式的软件架构：分布式计算的模式语言(卷4)［Ｍ］.北京：人民邮电出版社，2010：348．

［4］李林锋．分布式服务框架:原理与实践［Ｍ］.北京：电子工业出版社，2016：312．

［5］曾宪杰．大型网站系统与Java中间件实践［Ｍ］.北京：电子工业出版社，2014：340．

［6］Michael T. Nygard．Release It! Design and Deploy Production-Ready Software［Ｍ］.美国：The Pragmatic Programmers，2007：300．

［7］Sam Newman．Building Microservices：Designing Fine-Grained Systems 微服务设计［Ｍ］.北京：人民邮电出版社，2016：228．

［8］张开涛. 亿级流量网站架构核心技术——跟开涛学搭建高可用高并发系统[M]. 北京:电子工业出版社, 2017.

［9］布彻, (Paul, Butcher), (作者), 黄炎, (译者). 七周七并发模型[M]. 北京:人民邮电出版社, 2015.

［10］宸存, 巴斯, 克莱门茨,等. 软件构架实践(第二版)[M]. 清华大学出版社, 2004.

［11］Hamilton J. On designing and deploying internet-scale services[C] Conference on Large Installation System Administration Conference. USENIX Association, 2007:18.

［12］Mcheick H, Mohammed Z R, Lakiss A. Evaluation of Load Balance Algorithms[C] International Conference on Software Engineering Research, Management and Applications. IEEE, 2011:104-109.

［13］郭正敏. 基于SOA架构的分布式服务化治理方案的研究[D]. 南京邮电大学, 2016.

［14］章耿. 服务化框架技术选型与京东JSF解密[EB/OL]. http://mt.sohu.com/20161222/n476595246.shtml, 2016-12-22

［15］臧秀涛. 专访阿里巴巴毕玄：异地多活数据中心项目的来龙去脉[EB/OL]. http://www.infoq.com/cn/articles/interview-alibaba-bixuan, 2015-04-07

［16］Chris, Richardson. Microservices patterns[EB/OL]. http://microservices.io.

［17］Tony, Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design[EB/OL]. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015-02-19

［18］刘丁. 序列化和反序列化[EB/OL]. http://tech.meituan.com/serialization\_vs\_deserialization.html, 2015-02-26

［19］林世洪. 京东大促备战思路和方法2.0解密[EB/OL].http://zhuanlan.51cto.com/art/201612/525311.htm, 2016-12-19

［20］小邪. 稳定性思考-强弱依赖 [EB/OL]. http://jm.taobao.org/2012/10/31/stability-considerations-dependent-strength/.

［21］阿里小邪. 稳定性思考-强弱依赖2[EB/OL]. http://jm.taobao.org/2012/10/31/stability-considerations-dependent-strength-2/.

［22］Dubbo开发团队. Dubbo用户指南[EB/OL]. <http://dubbo.io/User+Guide-zh.htm>.

［23］绿麟. 服务容错模式[EB/OL]. <http://tech.meituan.com/service-fault-tolerant-pattern.html>.

［24］What is Akka?[EB/OL]. http://doc.akka.io/docs/akka/snapshot/intro/what-is-akka.html.

［25］Shailesh, Mangal. Monitoring Connect applications[EB/OL]. <https://developer.atlassian.com/blog/2015/08/monitoring-connect-applications/>.

［26］blogdevteam. Dubbo 团队成员梁飞专访 [EB/OL].http://blog.csdn.net/blogdevteam/article/details/8173233.

［27］梁宇鹏. 单元化架构，为什么要用以及我们如何做到[EB/OL]. <http://www.infoq.com/cn/articles/how-weibo-do-unit-architecture>.

［28］邱小侠. 微服务（Microservice）那点事[EB/OL]. <https://yq.aliyun.com/articles/2764>.

［29］Jonas, Boner. Scalability,Availability & Stability Patterns[EB/OL]. <http://www.mif.vu.lt/~donatas/Vadovavimas/Temos/Reactive%20systems/Scalability/2010%20Scalability%20patterns%28slides%29.pdf>.

［30］点评. RPC framework of DIANPING[EB/OL]. <https://github.com/dianping/pigeon>.

［31］陶明. 一种分布式服务框架的设计与实现[D]. 北京邮电大学, 2013.

# 致 谢

本文是在我的导师龚奕利教授的悉心指导下完成的。从课程的学习、论文的选题及最终定稿都得到了导师的帮助。导师严谨的治学态度，平易近人、诲人不倦的品格给我留下了深刻的印象，使我终生受益。在此，谨向导师表示最深深的敬意和由衷的感谢!

同时感谢武大计算机学院，感谢所有的授课和指导老师，对我在学业上的帮助和生活上的关心!正是你们孜孜不倦的传道、授业、解惑，使我在思想上和学业上都取得了巨大的进步。

感谢在这两年多的学习和生活中帮助、关心我的同学以及给我的论文提供许多宝贵素材的朋友们。有了这么多人的帮助，使我顺利的完成了我的研究生阶段学习，在今后的工作中，我会将这些人的帮助化为自己前进的动力。

最后，我要特别感谢我的家人长期以来对我默默的奉献和支持,是他们为我提供了坚忍不拔和不断进取的强大精神动力，使我能顺利完成学业。家是我终生的精 神寄托和心灵归宿，也是我今后人生道路上扬帆远航为理想打拼的起点。感谢我的父母为我营造了一个的温馨的港湾，当我身心疲弊时能得以憩息和修养。

在毕业之际衷心的祝愿你们生活美满、前程似锦!